Design and Implementation of an FPFA

M.Sc. Thesis

H. Bouma

Supervisors : Prof.Dr.Ir. C.H. Slump
Ir. J. Smit
Ir. P.M. Heysters
Report code : EL-S&S-006N01
Student number : 9503188
Date : July 3, 2001
Period : September 2000 - July 2001

University of Twente
Department of Electrical Engineering
Laboratory of Signals and Systems

Abstract

Future hand-held multi-media devices require high performance on a small energy budget and short time-to-market. In the *Chameleon* project, a reconfigurable architecture is designed to solve these issues for digital-signal-processing algorithms. This thesis describes an array of reconfigurable processors – the *Field Programmable Function Array (FPFA)* – and how it is designed and implemented within this project. The FPFA architecture is specified in high-level, but synthesizable VHDL, with a full-custom chip implementation in mind.

A demonstrator will be built on a Xilinx Virtex FPGA to verify the FPFA concept, therefore synthesis is done with FPGA Express. A realistic application (FFT) has been simulated. Speed and chip area are estimated.

‘Church of Reason’ lecture

The real University, he said, has no specific location. It owns no property, pays no salaries and receives no material dues. The real University is a state of mind. It is that great heritage of rational thought that has been brought down to us through the centuries and which does not exist at any specific location. It’s a state of mind which is regenerated throughout the centuries by a body of people who traditionally carry the title of professor, but even that title is not part of the real University. The real University is nothing less than the continuing body of reason itself.

Robert M. Pirsig, Zen and the Art of Motorcycle Maintenance.

Acknowledgements

This thesis concludes my last year of Electrical Engineering at the University of Twente. I could not have done this project alone, therefore I want to thank many people for their support.

First of all, I would like to thank Paul Heysters, who really took a lot of time to help me. Not just for writing his comments on the report writing – which he did thoroughly – but also for his guidance and advice through the year, and I want to thank Jaap Smit for all the discussions we had about low-power design and the FPFA.

Also thanks to the people of the S&S-NT group for their support and for the good atmosphere, and to the people in the Chameleon project for the meetings.

And last, but not least, I want to thank Jos Huisken and Vishal Choudhary (Philips Research) for the timing and area estimations in a CMOS18 process.

Enschede, June 2001

Henri Bouma

Contents

1 Introduction **11**
1.1 Background . . . 11
1.2 Organization of the Report . . . 12

2 Analysis of the FPFA Design **13**
2.1 Field Programmable Function Array . . . 13
2.2 Number System . . . 14
2.2.1 Binary Integer Numbers . . . 14
2.2.2 Binary Fractional Numbers . . . 14
2.3 Arithmetic and Logic Unit . . . 15
2.3.1 Level One . . . 15
2.3.2 Level Two . . . 17
2.3.3 Level Three . . . 17
2.4 Register File . . . 17
2.5 Crossbar Switch . . . 18
2.6 Memories . . . 19
2.7 Conclusion . . . 19

3 DSP Algorithms **21**
3.1 Linear Interpolation . . . 21
3.1.1 Interpolation Algorithm . . . 21
3.1.2 Interpolator Mapping . . . 22
3.2 Finite Impulse Response . . . 24
3.2.1 FIR Algorithm . . . 24
3.2.2 FIR Mapping . . . 26
3.3 Fast Fourier Transform . . . 26
3.3.1 FFT Algorithm . . . 26
3.3.2 FFT Mapping . . . 28
3.4 Conclusion . . . 29

4 Implementation of the FPFA **31**
4.1 Datapath . . . 31
4.1.1 Arithmetic and Logic Unit . . . 32

4.1.2 Registers 33
4.1.3 Crossbar Switch 35
4.1.4 Memory and Counter 37
4.2 Control 40
4.2.1 Configuration Registers 40
4.2.2 Vertical Microprogramming 42
4.2.3 Decoder 43
4.2.4 Tile Control 44
4.2.5 External Control 49
4.3 Evaluation 53
4.4 Conclusion 56

5 Synthesis **57**
5.1 Foundation Series 57
5.1.1 Project Manager 57
5.1.2 Tools 58
5.2 Virtex FPGA 59
5.3 Logic synthesis 61
5.4 Results 62

6 Simulation **65**
6.1 Functional Simulation 65
6.2 Timing Simulation 67
6.3 Test Bench 68
6.3.1 Test Bench 69
6.3.2 Test Package 69
6.3.3 Command File 71
6.4 Results 72

7 Conclusions and Recommendations **73**
7.1 Conclusions 73
7.2 Recommendations 74

Bibliography **75**

A Arithmetic and Logic Unit **77**

B Configuration Registers **79**

C Macro for Simulation **81**
C.1 Functional Simulation 81
C.2 Timing simulation 82

Abbreviations

General used abbreviations

3D	Three dimensional
ABS	Absolute value
ALU	Arithmetic and logic unit
ASIC	Application specific integrated circuit
BIN	Binary
CHAR	Character
CLB	Configurable logic block
DEC	Decimal
DFT	Discrete Fourier transform
DSP	Digital signal processing
DSP	Digital signal processor
FFT	Fast Fourier transform
FIFO	First in first out
FIR	Finite impulse response
FPFA	Field programmable function array
FPGA	Field programmable gate array
FT	Fourier transform
GPP	General purpose processor
HEX	Hexadecimal
INT	Integer
IO, I/O	Input-output
IOB	Input-output block
LIFO	Last in first out
LUT	Look-up table
MSB	Most significant bit
RAM	Random access memory
STR	String
VHDL	VHSIC hardware description language

Subscript abbreviations

fp	Fixed point
H	High; Most significant part
im	Imaginary; Complex imaginary part
L	Low; Least significant part
re	Real; Complex real part
se	Sign extended

Common signal or entity abbreviations

addr	Address
clk	Clock
cnt	Counter
CR	Configuration register
di/din	Data input
dir	Direction
do/dout	Data output
en	Enable
ld	Load
mem	Memory
mux	Multiplexer
rd	Read
reg	Register
rst	Reset
sl	Standard logic
slv	Standard logic vector
xbar	Crossbar
we	Write enable
wr	Write

Chapter 1

Introduction

1.1 Background

The most flexible processing architectures are *general purpose processors* (GPPs), including the large class of *digital signal processors* (DSPs). To achieve performance for a wide range of applications, GPPs dedicate a substantial amount of die area to overhead such as speculative execution and branch prediction. These complex mechanisms can extract a moderate amount of parallelism but not the large amount available in many compute-intensive applications.

At the other end of the flexibility spectrum are the *application specific integrated circuits* (ASICs) which can be used to achieve higher performance at lower area and energy cost than GPPs. High performance can be achieved since the architecture can be tailored for a specific application to extract the parallelism, while optimizing for power, speed or area. However the drawback of ASICs are their lack of flexibility, their high design cost and high design time. This makes them unattractive except for very well-defined and wide-spread applications.

Field programmable gate arrays (FPGAs) promised to bridge the flexibility and performance gap between GPPs and ASICs. FPGAs are more flexible than ASICs and they can exploit the parallelism in algorithms better than GPPs. Unfortunately, to implement arbitrary circuits, FPGAs have to be very fine-grained (the logic blocks must be small and regular) and the overhead of this generality can be expensive in both area and performance. While GPPs use highly optimized functional units that operate on long data words, FPGAs are only efficient for complex bit-oriented computations or complicated bit-level masking and filtering [Cro99].

In the Chameleon project [Cha99], the opportunities of reconfiguration of a mobile multi-media system are studied. A reconfigurable architecture might be the key to flexible low-power hand-held systems. A novel as-

pect in this concept is that chip design is replaced by dynamic reconfiguration for future applications using a highly efficient platform. This sheds a new light on the fundamental issues of low-power embedded systems. The *Field programmable function array* (FPFA) is optimized for highly repetitive, computationally-intensive DSP (digital signal processing) algorithms. It has a reconfigurable datapath to create more flexibility than in the ASIC and it has shorter design times because execution of another specific application does not require the design of a new integrated circuit. Many computations can be performed in parallel, therefore the performance is much higher than GPPs and because the datapath has a low power radius [Hey00] it is more energy efficient than GPPs and FPGAs.

1.2 Organization of the Report

In Chapter 2 the concept of an FPFA will be explained and the available literature on the initial design and optimizations of the FPFA is analyzed. The FPFA is optimized for DSP algorithms, therefore Chapter 3 will explain some of these algorithms and how they can be mapped on the design of the FPFA. The mapping exercises provide insight in the required capabilities of the FPFA. Based on these new insights, an improved and more complete design of the FPFA is proposed in Chapter 4. A demonstrator will be built on a Xilinx Virtex FPGA to verify the FPFA concept. Therefore, Chapter 5 considers the Virtex and the Xilinx tools which are used for synthesis. ModelSim, the tool used for functional and timing simulation, is described in Chapter 6. These two chapters are about the tools and the results of the tools. Finally, Chapter 7 discusses the conclusions and recommendations.

Chapter 2

Analysis of the FPFA Design

2.1 Field Programmable Function Array

The *Field programmable function array* (FPFA) is a low power, reconfigurable accelerator for DSP algorithms. High performance is obtained by exploiting parallelism. Custom hardware could be the solution for fast and efficient calculations, however design cycles of custom hardware are expensive and algorithms realized in hardware can not be changed. Analogous with the *Field programmable gate array* (FPGA), an FPFA architecture contains a repeated structure, with operators, memory and programmable interconnection points. As opposed to FPGAs, the FPFA uses w bits wide (for example 20 bits wide) operations and busses. The operators in an FPFA are mainly arithmetical, while the operators in an FPGA are logical. With this structure, the FPFA combines the advantage of implicit parallelism of an FPGA with the arithmetical richness of a microprocessor. Targets of the FPFA are computation intensive real-time applications. The FPFA has a regular matrix of processor tiles (Figure 2.1), which consist of five ALUs, registers, interconnect and memories.

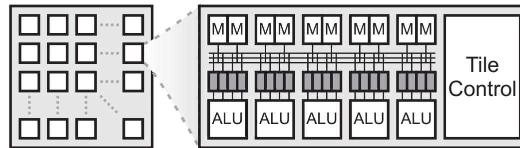

Figure 2.1: FPFA architecture

In this chapter an overview of the evolution of the FPFA architecture is given as proposed in [Rem97], [Kla98], [Ste99], [Ash00] and [Hey00]. The old designs are analyzed in order to make a redesign and implementation possible. Before this analysis, the theory about binary numbers is summarized.

2.2 Number System

2.2.1 Binary Integer Numbers

There are many ways to represent integers as bit patterns (Table 2.1). The *unsigned binary code* is for nonnegative integers. For code of length n, the 2^n patterns represent integers 0 through $2^n - 1$.

Sign-magnitude code is used for integers, both positive and negative (range $-2^{n-1} + 1$ through $2^{n-1} - 1$). The sign bit (most significant bit, MSB) is 0 for positive and 1 for negative numbers; the other bits carry the magnitude. Changing from positive to negative numbers with this representation is more energy efficient than with two's complement, since less bits will change their value.

Two's complement code is also used for positive and negative integers. For code of length n, they represent integers -2^{n-1} through $2^{n-1} - 1$. Negative integers are formed by complementing each bit of the corresponding positive integer and incrementing the integer by one. Negative integers of *one's complement* are obtained by complementing each bit of the corresponding positive integer without incrementing.

Int	Binary Code	Sign Magnitude	2's Complement	1's Complement	Int
	unsigned [0,7]	signed [-3,3]	signed [-4,3]	signed [-3,3]	
-4			1 0 0		-4
-3		1 1 1	1 0 1	1 0 0	-3
-2		1 1 0	1 1 0	1 0 1	-2
-1		1 0 1	1 1 1	1 1 0	-1
0	0 0 0	0 0 0	0 0 0	0 0 0	0
1	0 0 1	0 0 1	0 0 1	0 0 1	1
2	0 1 0	0 1 0	0 1 0	0 1 0	2
3	0 1 1	0 1 1	0 1 1	0 1 1	3
4	1 0 0				4
5	1 0 1				5
6	1 1 0				6
7	1 1 1				7

Table 2.1: Three bit integer codes

2.2.2 Binary Fractional Numbers

Fractions present another number representation problem. How can one represent fractional quantities using bits? Representing fractions can be

done using fixed-point or floating-point numbers. Fixed-point numbers are represented in the same way as representing integers (Table 2.2). The rules for binary arithmetic on fixed-point numbers are the same as for integers, so it requires no change in the hardware to deal with binary fractions. Two numbers can only be added meaningfully if the binary point is aligned. Of course, the arithmetic unit will add them no matter where you think the binary point is. This is just like decimal arithmetic, the decimal point must be aligned to add two numbers. On one hand, fixed point numbers are ideal for applications where efficiency and precision are important and all the numbers lie within a limited range. On the other hand, floating point numbers are more flexible, they are coded as a fraction multiplied by an exponent. The implementation of floating-point numbers is more complex than the implementation of fixed-point. More energy is used to perform the calculations, more hardware is needed for the implementation and most DSP calculations can be done in fixed-point notation. Therefore fixed-point calculations are used in the FPFA.

Decimal	3 integer 5 fraction	6 integer 2 fraction
4.00	1 0 0.0 0 0 0 0	0 0 0 1 0 0.0 0
2.00	0 1 0.0 0 0 0 0	0 0 0 0 1 0.0 0
1.00	0 0 1.0 0 0 0 0	0 0 0 0 0 1.0 0
0.50	0 0 0.1 0 0 0 0	0 0 0 0 0 0.1 0
0.25	0 0 0.0 1 0 0 0	0 0 0 0 0 0.0 1

Table 2.2: Fixed-point binary numbers

2.3 Arithmetic and Logic Unit

The datapath of the arithmetic and logic unit (ALU) is depicted in Figure 2.2. The ALU has four data inputs (a, b, c and d) and two data outputs (out1 and out2). The in- and outputs are 20-bit wide and should use a sign-magnitude representation because this is more energy efficient. Input *east* and output *west* (40-bit wide) connects neighboring ALUs. In the ALU three different levels can be discriminated.

2.3.1 Level One

Level one is a reconfigurable function block. Each function f_1, f_2 and f_3 in Figure 2.2 returns the following result.

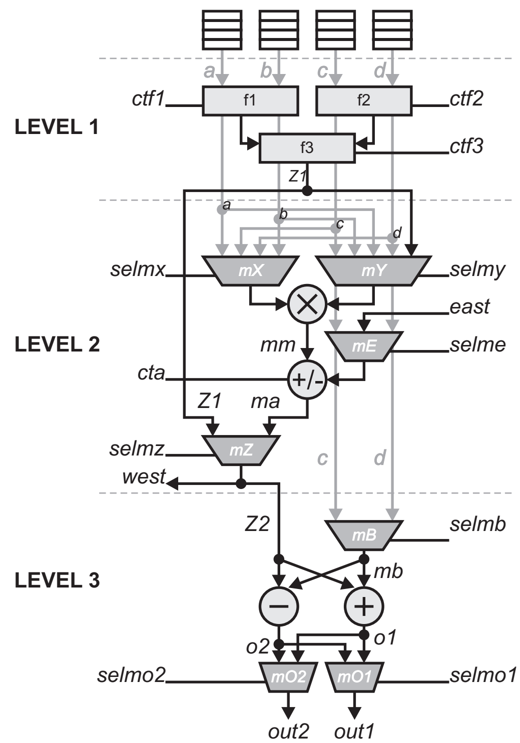

Figure 2.2: ALU datapath (Appendix A)

$$\begin{aligned} f_n = 0 \\ &| \ [abs][-]Op_{left} \\ &| \ [abs][-]Op_{right} \\ &| \ [abs] + ([-]Op_{left}, [-]Op_{right}) \\ &| \ [abs]min([-]Op_{left}, [-]Op_{right}) \\ &| \ [abs]max([-]Op_{left}, [-]Op_{right}) \end{aligned} \tag{2.1}$$

$$Z1 = f_3(f_1(a, b), f_2(c, d)) \tag{2.2}$$

2.3.2 Level Two

Level two contains a 19*19 bit unsigned multiplier and a 40 bit wide adder. The *east-west interconnect* connects neighboring ALUs without using registers (Section 2.4) and the crossbar (Section 2.5). (The *se* subscript means that the numbers are sign extended.)

$$\begin{aligned} Z2 = Z1 \\ &| \ + \Big((a|b|c|d) * (a|b|c|d|Z1), [-](0|c_{se}|d_{se}|east)\Big) \end{aligned} \tag{2.3}$$

2.3.3 Level Three

Level three can be used as a 40-bit wide adder or as a butterfly structure.

$$\begin{aligned} o1 &= +(0|c_{se}|d_{se}|c\&d, Z2) \\ o2 &= +(0|c_{se}|d_{se}|c\&d, -Z2) \end{aligned} \tag{2.4}$$

$$\begin{aligned} out1 &= o1_{high}|o1_{low}|o2_{high}|o2_{low} \\ out2 &= o1_{high}|o1_{low}|o2_{high}|o2_{low} \end{aligned} \tag{2.5}$$

2.4 Register File

Registers can be used for storing constant numbers and temporary values [Rem97]. They are used to speed up the maximum clock frequency and to avoid extra power consumption over long wires outside the ALU.

Instead of registers, a multi-port register file can be used, which contains a number of registers and a (possibly different) number of input and output

ports. This structure is very flexible, every input can write to every register and every output can read from every register. Büchli [Büc98] and Marsman [Mar98] both advise a bank with eight registers in front of the ALU and a bank with four registers behind the ALU. The flexible structure with a minimum delay of one clock cycle can also be used to store static inputs or to create delays of several clock cycles [Ste99]. Ashtari [Ash00] proposes four register banks and for each bank four registers. So there are sixteen registers available for each ALU. The number of registers – which is a trade off between size, energy and flexibility – depends on the DSP algorithms. Therefore, more information about the registers is given (after the chapter about algorithms) in Subsection 4.1.2.

Latches

Finite delays in static logic designs can result in dynamic hazards. Hence, the multiplier can have transitions in one clock cycle before stabilizing to the correct level. Latches at the inputs of the multiplier can be used to prevent the multiplier from consuming much energy [Ste99].

2.5 Crossbar Switch

The crossbar is a row of horizontal busses, crossed by vertical busses (Figure 2.3). It makes the flexible routing between five ALUs and ten memories possible. The crossbar contains six horizontal busses, connecting the vertical busses. Besides the six horizontal busses four extra local lines are added, two for the outputs of one ALU and two for the outputs of the two memories. This way each ALU can be connected with its own memories without using one of the busses and the data can easily be stored locally. These small local busses are energy efficient and prevent more complex bus routing, because each ALU can read from and write to the memories and read its own outputs without needing one of the six horizontal busses.

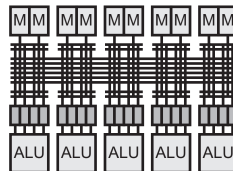

Figure 2.3: Crossbar switch

2.6 Memories

Memories in the FPFA tile can be used for storing arithmetical data at discrete positions (like a sine function), input and output buffers and buffers for temporary data. Each ALU has two data memories. The number of entries of the memories is a trade off between the number of needed reloads to complete an algorithm, the size on the chip of the memory and the power consumption of the memory. There is a trade off, because a small memory needs more reloads than a large memory for completing an algorithm, but it has a lower power consumption. The memories have 256 entries, which are just enough for reasonably accurate LUTs [Ste99].

2.7 Conclusion

In this chapter an overview was given of the FPFA architecture as proposed by others. The ALU should consist of logical and arithmetical functions in the first level, a multiplier and adders/substractors in the second and some adders and substractors in the last level. The registers are used for storage and for pipelining the ALUs, the crossbar for interconnect and the memories for storage of results or tables. Before redesigning the architecture, we will look what implementation is required by the DSP algorithms.

Chapter 3

DSP Algorithms

Digital signal processing (DSP) is the theory associated with processing digital signals. Signals can be continuous-time or discrete-time signals depending on whether the independent variable, time, takes on a continuum or a set of discrete values, respectively. Discrete-time signals are obtained by sampling a continuous-time signal although they may arise naturally in other contexts. In practical applications of digital signal processing, we use digital signals, which are quantized both in amplitude and in time. Amplitude quantization is necessary, if digital devices are used to store and process the information. The design of the FPFA is derived from several often used algorithms. These are only a few of the algorithms that it should be able to handle. The algorithms considered are the linear interpolation, the finite-impulse response (FIR) filter and the fast Fourier transform (FFT). Each section of this chapter discusses one of these algorithms and shows the mapping of the algorithm onto the FPFA.

3.1 Linear Interpolation

3.1.1 Interpolation Algorithm

A continuous function described with a set of discrete samples, can be approximated with an interpolation function. Often, small tables are used to handle difficult calculations like square root, division or a sine. Therefore, it might be useful to interpolate between the values from the table. Reconstruction of the function is possible with an ideal interpolator if the function is band limited to half of the sample frequency. The main problem of an ideal interpolator is that the pulse response is infinitly long. To solve this problem, windowing techniques are often used. However, this is not a good solution for a real-time interpolation formula, because we want a formula with an extreme low computational complexity. Therefore linear interpolation is suggested for the FPFA [Rem97]. Only two sampled values are needed to calculate an intermediate value (Figure 3.1) and the

computational complexity of the linear interpolation algorithm is just one multiplication (Equation 3.1 and Figure 3.2).

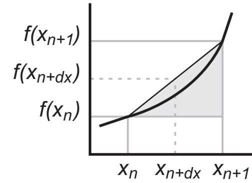

Figure 3.1: Linear interpolation plot

$$f(x_{n+dx}) = \Big(f(x_{n+1}) - f(x_n)\Big) * dx + f(x_n) \tag{3.1}$$

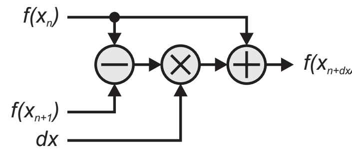

Figure 3.2: Linear interpolation algorithm

Multiple linear interpolators can be used to interpolate in 2D or 3D [Ste99]. In three dimensions, seven linear interpolators are used, value dx is routed to the first column of four interpolators, dy to the second column of two interpolators and the value dz to the last interpolator in the third column (Figure 3.3). Eight sampled values are needed to calculate an intermediate value. The first column is used to reduce the cube of eight points to a plane, the second is used to reduce the plane of four points to a line and the third row is used to reduce the line of two points to one point (Figure 3.4).

3.1.2 Interpolator Mapping

In this subsection, an implementation of the linear interpolator on an FPFA is given.

In Figure 3.5a and Equation 3.2 the datapath of a linear interpolator, mapped onto the FPFA ALU (Figure 2.2) is given. The signals in the equation are referring to the signals described in Section 2.3. To calculate a point between two values of x (for example x_n and x_{n+1}), both values must be present at the same time. One way to be able to read two values of x at the same time is to store even and odd values in different memories. To calculate a value between x_n and x_{n+1}, two values can be read without

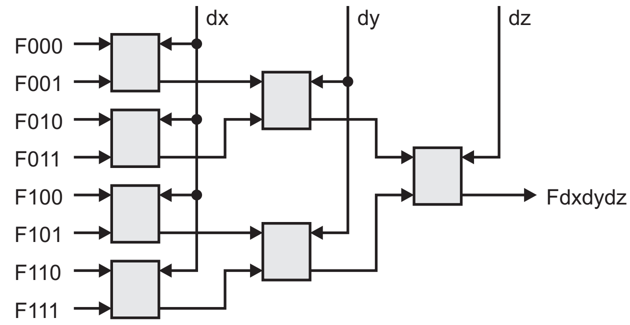

Figure 3.3: 3D linear interpolation algorithm

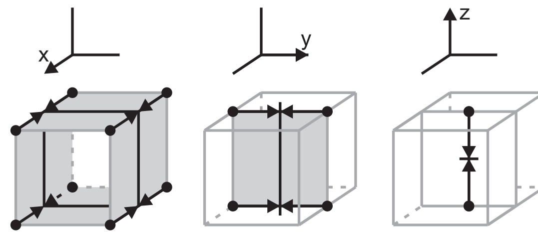

Figure 3.4: 3D linear interpolation plot

a swap (Fig. 3.5b) and to calculate a value between x_{n+1} and x_{n+2}, the operands have to be swapped (Fig. 3.5c). The crossbar can be used to swap the even and odd operands or the reconfigurable function blocks (in level one of the ALU) can be used to swap the operands. Another way to be able to read two values of x at the same time is to insert a delay buffer (Fig. 3.5d). One of the inputs of the ALU will read x_{n+1} and the other will read the delayed value x_{n+2}.

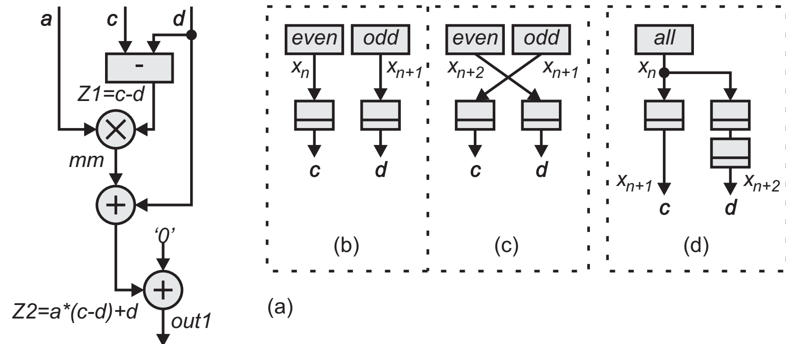

Figure 3.5: Implementation of linear interpolation (a) ALU implementation (b, c) Store even and odd values in different memories (d) Insert delay buffer

$$
\begin{aligned}
Z1 &= c - d \\
Z2 &= a * Z1 + d \\
O1 &= Z2
\end{aligned}
\tag{3.2}
$$

Seven linear interpolators are needed to interpolate in three dimensions. Because one tile contains only five ALUs and each interpolator needs an ALU, the 3D interpolation cannot be performed every clock cycle by one tile. The four interpolators in the first column (Figure 3.3) can be calculated in parallel with the interpolator in the third column, as shown in Figure 3.6. Therefore, by pipelining every two cycles one 3D interpolation can be performed.

3.2 Finite Impulse Response

3.2.1 FIR Algorithm

The *Finite impulse response* (FIR) filter is a frequently used algorithm in digital signal processing applications. FIR filters are useful in applications where frequency dispersion effects caused by nonlinear phase response must

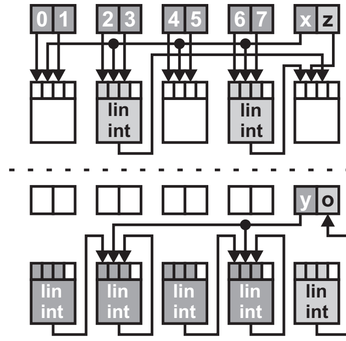

Figure 3.6: Implementation of 3D interpolation

be minimized, such as data transmission. Another advantage of these filters is the stability and the simple analysis [Rob87]. The FIR filter can easily be configured as a low-pass, high-pass, band-pass or band-stop filter by simply adjusting the filter coefficients. Each clock tick, a new value is shifted in the filter and the resulting new state is used to compute all the filter values in parallel [Kla98]. Figure 3.7 shows two 4-tap FIR filters, both according to Equation 3.3, which can be converted to each other with a transformation.

$$y(k) = \sum_{i=0}^{n-1} b_i * x_{k-i} \tag{3.3}$$

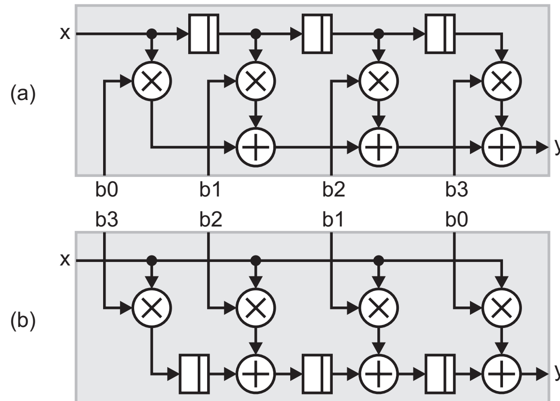

Figure 3.7: Two 4-tap FIR filters

3.2.2 FIR Mapping

Each section (tap) of the FIR filters in Figure 3.7 contains a multiplication, an addition and a delay. Therefore, five taps can be mapped on one tile with five processing parts. The taps in the second filter (b) contain less outputs than the tabs in the first filter (a) and another advantage is that the longest path is shorter [Ste99]. Therefore, the second filter is used to implement the FIR algorithm (Figure 3.8).

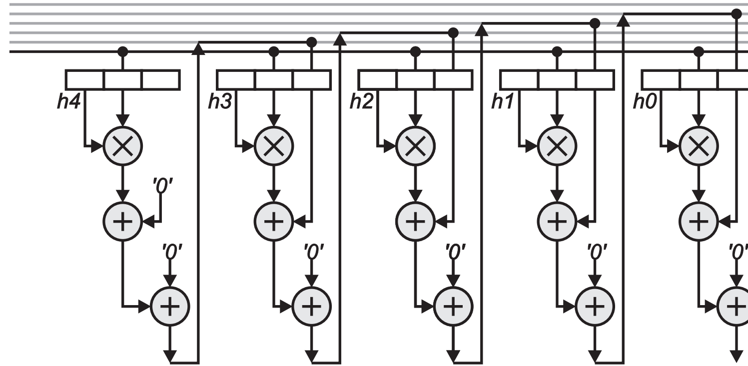

Figure 3.8: Mapping of FIR filter

To perform a 5-tap FIR filter efficiently, at least six crossbar busses are needed for an FPFA tile. For each processing part, three registers are used. One for storing a constant number, one for the input and one for the result of the previous tap. Each processing part needs one adder and one multiplier.

3.3 Fast Fourier Transform

3.3.1 FFT Algorithm

The Fourier transform (FT) transforms a signal from the time to the frequency domain and the inverse FT vice versa. For digital signal processing, we are particularly interested in the *Discrete Fourier transform* (DFT). The *Fast Fourier transform* (FFT) can be used to calculate a DFT efficiently.

How many computations are required? Suppose the number of data points is a power of 2. The idea behind the FFT is to break up the original N point sample into two sequences. This is because a series of smaller problems is easier to solve than one large one. For a N-point DFT N^2 computations are required, however a N/2-point DFT requires only $(N/2)^2 = (N^2)/4$ computations. The N-point DFT can be divided in one N-point FFT (using N calculations) and two N/2-point DFTs [Rob87]. This reduces the number of calculations to $N + 2 * (N^2)/4 = N + (N^2)/2$. Hence, since N is a

power of 2, we can repeat the decimation process and reduce the number of computations to $Nlog_2(N)$ multiplications (Figure 3.9).

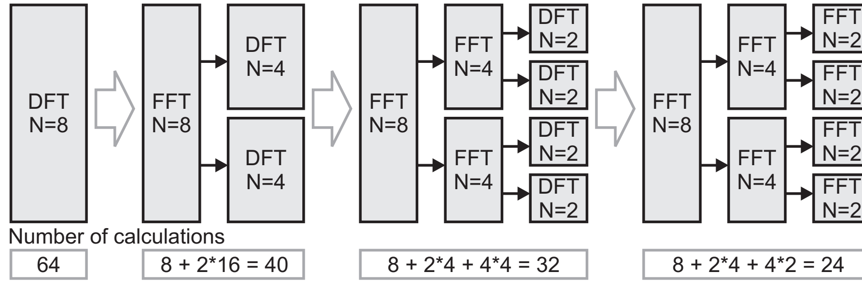

Figure 3.9: Use FFT-2 recursively to calculate DFT efficiently

The operations needed to perform a DFT can be rewritten to a simple structure called a radix-2 FFT butterfly (Equation 3.4 and Figure 3.10), which is repeated many times with different inputs. Inputs are written in small case (a, b), outputs in capitals (A, B) and the twiddle factors are called W.

$$\begin{aligned} A &= a + W * b \\ B &= a - W * b \end{aligned} \tag{3.4}$$

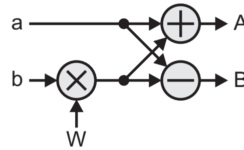

Figure 3.10: Radix-2 FFT butterfly

The complex FFT-2 butterfly has six inputs (a_{re}, a_{im}, b_{re}, b_{im}, W_{re} and W_{im}) and four outputs (A_{re}, A_{im}, B_{re}, B_{im}) (Equations 3.5, 3.6 and 3.7 and Figure 3.11).

$$\begin{aligned} A &= a_{re} + j * a_{im} + (W_{re} + j * W_{im}) * (b_{re} + j * b_{im}) \\ B &= a_{re} + j * a_{im} - (W_{re} + j * W_{im}) * (b_{re} + j * b_{im}) \end{aligned} \tag{3.5}$$

$$\begin{aligned} A &= a_{re} + j * a_{im} + W_{re} * b_{re} + j * W_{im} * b_{re} + W_{re} * j * b_{im} - W_{im} * b_{im} \\ B &= a_{re} + j * a_{im} - W_{re} * b_{re} - j * W_{im} * b_{re} - W_{re} * j * b_{im} + W_{im} * b_{im} \end{aligned} \tag{3.6}$$

$$A_{re} = a_{re} + W_{re} * b_{re} - W_{im} * b_{im}$$

$$A_{im} = a_{im} + W_{im} * b_{re} + W_{re} * b_{im}$$
$$B_{re} = a_{re} - W_{re} * b_{re} + W_{im} * b_{im}$$
$$B_{im} = a_{im} - W_{im} * b_{re} - W_{re} * b_{im} \quad (3.7)$$

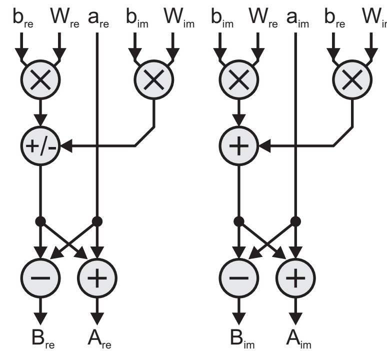

Figure 3.11: Mapping of radix 2 FFT

3.3.2 FFT Mapping

To perform a DFT efficiently, the FFT-2 butterfly (Figure 3.10) can be performed recursively. Four ALUs, ten memories and six busses are used to implement the complex radix-2 butterfly (Figure 3.12).

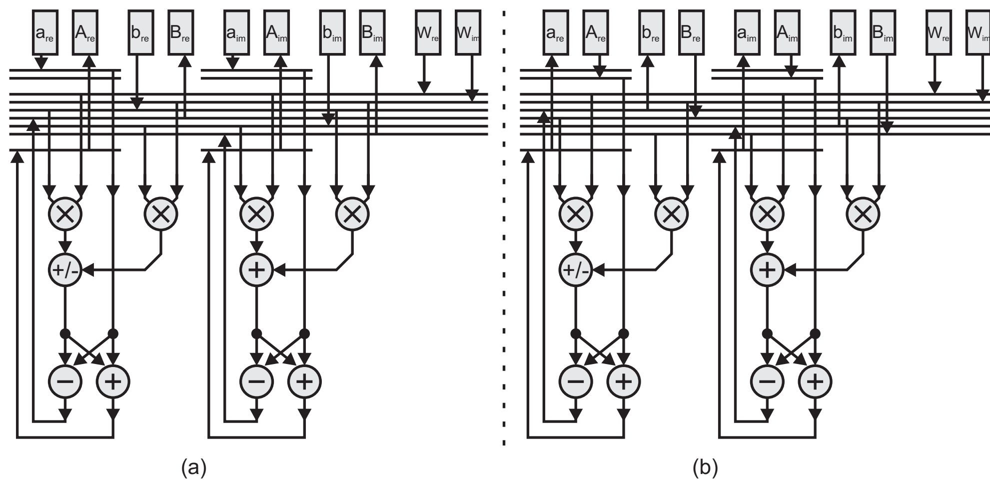

Figure 3.12: Calculation of FFT (a) odd column (b) even column

When the number of computations is reduced by the decimation process (Figure 3.9) until every calculation can be executed by a radix-2 FFT butterfly, the algorithm is as shown in Figure 3.13. For a 16-point DFT, $16 * log_2(16) = 64$ multiplications are required. To perform a radix-16 FFT, 32 complex radix-2 FFT must be executed one by one on an FPFA tile.

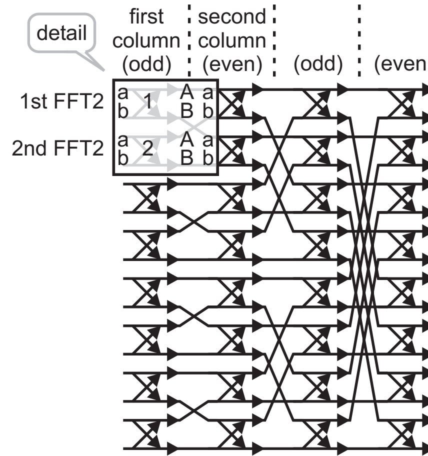

Figure 3.13: Recursive use of FFT-2; writing to same memory

After eight calculations, the first column of Figure 3.13 is executed. First, the first FFT-2 reads its input values (a, b) from the memories and writes its output values (A, B) to the memories, then the second, and so on (Figure 3.12a). After the first column, each of the FFTs in the second column will take its input values from the Memories A and B and write its results to the Memories a and b (Figure 3.12b). However, the first column will have to put its results in the correct memory, in order to make it possible for the second column to execute a FFT-2 every clock cycle. Hence, as we can see in the detail of Figure 3.13, both outputs (A and B) of the first FFT will have to be written to Memory a and both outputs of the second FFT will have to be written to Memory b. Since it is not possible to write two values to the same destination at the same moment, the architecture should be changed to allow a butterfly with twist (Subsection 4.1.2).

3.4 Conclusion

Mapping of the linear interpolation, FIR filter and FFT onto an FPFA have been shown. Of course, the FPFA should be able to perform many more operations. These algorithms have been chosen to find a representative set of the requirements for the implementation of the FPFA.

Chapter 4

Implementation of the FPFA

The design of an FPFA tile consists of two main parts: datapath and control (Figure 2.1, page 13). The fetching and interpretation of instructions is done in the instruction unit or control unit. The unit that performs the operations, such as addition or multiplication, is the execution unit. The part of the execution unit in which data are transported and transformed is called the datapath. The central part of the datapath is called the *arithmetic and logic unit* (ALU).

Two modes can be distinguished: operation and control mode. In normal operation mode, the ALU can operate normally, data can be processed and stored into memories and registers. The control mode is used to store system information in the control parts. The datapath is described in the first section of this chapter. The remaining part of the design serves to control the datapath. This control mechanism is discussed in the second section. The third section is used for an evaluation of the design.

4.1 Datapath

An FPFA consists of tiles, each tile consists of five processing parts and each processing part contains an ALU, registers, a crossbar and two memories (Figure 2.1). The FPFA design has two w-width data busses, one for incoming data (di) and one for outgoing data (do). The bus for incoming data can be connected to the first bus of the crossbar of a tile and the other can be connected to the second bus. In normal operating mode, the ALU can communicate with the memories and registers through the crossbar switch. Information about ALU, crossbar switch, memory and counter can be found in this section.

4.1.1 Arithmetic and Logic Unit

Much information about the ALU design is already in Section 2.3. However, the design of the ALU in subsection 2.3.3 is not able to calculate with fixed-point calculations. Therefore, the design of level 3 has changed.

Fixed-Point Architecture

Many digital-signal-processing algorithms calculate, not only with integers, but also with fixed- or floating-point numbers. A design for fixed-point calculations uses less hardware resources and it consumes less energy than floating-point calculations. Therefore, a fixed-point architecture is chosen (Section 2.2). Assumed is that the word width w is 20, with 15 numbers behind the point.

In sign-magnitude representation, this leaves four bits of data and one sign bit in front of the point. After a multiplication in single precision, the result will be in double precision (Figure 4.1) and bit 38 and 39 both are sign bits. When going from double precision to single precision, the bits 38 and 33-15 should be used to make a fixed-point number in normal precision.

For a two complement notation, which is used in the FPFA, the number of bits before the point is five in single precision and ten in double precision.

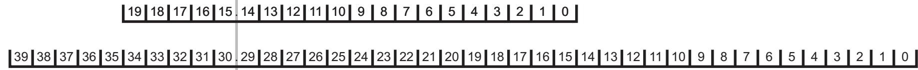

Figure 4.1: Fixed-point numbers (a) single (normal) precision (b) double precision

A fixed-point implementation was chosen. Therefore, level three of the ALU had to change. The equations in Subsection 2.3.3 are replaced by the Equations 4.1 and 4.2.

$$\begin{aligned} o1 &= +(0|c_{se}|c\&d|c_{fp}, Z2) \\ o2 &= +(0|c_{se}|c\&d|c_{fp}, -Z2) \end{aligned} \tag{4.1}$$

$$\begin{aligned} out1 &= o1_{fp}|o1_L|o1_H|o2_L \\ out2 &= o2_{fp}|o2_L|o2_H|o1_H \end{aligned} \tag{4.2}$$

A full description of the ALU data signals and control signals is given in Appendix A.

4.1.2 Registers

Registers can be used for pipelining, storing constant numbers and temporary values. How many registers are needed? Just one for each input (four) or also one for each output (six), eight in front of the ALU and four behind it or even four for each input (sixteen) (Section 2.4)? What number of registers is needed to map the algorithms efficiently on the FPFA and what structure should be used? To make a good decision, the constraints imposed by the DSP algorithms will be analyzed.

DSP Algorithms

- *Linear interpolation* uses three inputs, each with a register. To perform the three dimensional interpolation, no extra registers are needed (Figure 3.6).
- The *FIR filter* uses three register for each ALU (filter tap). One register for each input of the ALU is needed. This algorithm uses no register at the output of the ALU (Figure 3.8).
- The *FFT* algorithm requires at least one register at the output of the ALU to perform a twist, as already mentioned in Subsection 3.3.2.

FIFO Buffer

Besides pipelining and storing constant numbers and temporary values, registers can be used as delays in a DSP algorithm. Figure 4.2 shows an implementation that easily stores four constants and temporary values, that can be used for pipelining and to implement a variable length FIFO (first in first out) buffer [Cro99]. The buffer can store constants by disabling the write-enable signal after filling. The FIFO can be configured as 1 to 4 register delays, using static control signals. This is an advantage, because less configurations are needed and less control signals will have to change their value. This implementation is not used, because it is less flexible than a register file.

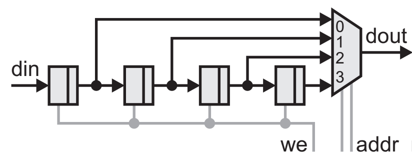

Figure 4.2: FIFO registers at input of ALU

Register File

Register files are very flexible. They are used by microprocessors and DSPs and register files can have multiple read and write ports. This implementation of the FPFA uses one register file for each input of the ALU. Each consists of four registers [Smi00], which can be read by one input and written by one output (Figure 4.3). Input and output ports can select the same register on the same moment. The register file can be used for pipelining and as multiplexer and it is suitable for low-power applications [Bla76].

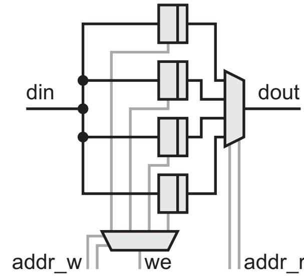

Figure 4.3: Register file

Butterfly with Twist

While calculating radix-2 FFTs, it is not always possible to write both outputs of an ALU in the correct memory. As a result, data will not be in place to perform the next radix-2 FFT. Therefore, it is not possible to calculate a radix-2 FFT every clock cycle unless the architecture is changed. One solution would be to use extra clock cycles to rearrange the data, however this makes the FPFA much slower. Another solution would be to change the architecture to allow a butterfly with twist (Figure 4.4).

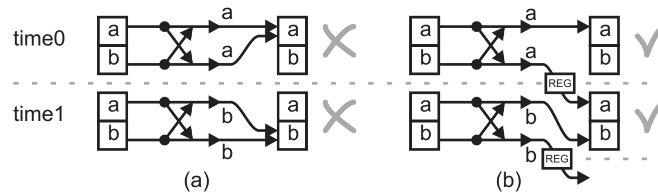

Figure 4.4: FFT butterfly writing in memories (a) without twist (b) with twist

At the outputs of the ALU, a smaller unit is used, consisting of one register that can be bypassed (Figure 4.5). This unit allows the implementation

of zero and one delay which is necessary to perform the FFT twist; more general, it is very useful when both outputs of the ALU have the same destination.

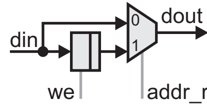

Figure 4.5: Register at output of ALU

Conclusion

Every ALU has two registers at its outputs – which can be bypassed – and four register files at its inputs (Figure 4.6). Each register file contains four registers.

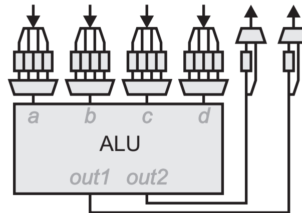

Figure 4.6: Registers at input and output of ALU

4.1.3 Crossbar Switch

The crossbar switch is a row of horizontal busses, crossed by vertical busses (Figure 4.7). This way ALUs and memories can be connected with each other.

The big advantage of this design is the flexibility at the inputs. The horizontal busses, connecting the five ALUs of one FPFA tile, take much chip area. Therefore, only six busses are used, this is the minimum needed to perform the algorithms (FIR and FFT) efficiently. Extra flexibility is added by using local busses for the two memories and ALU outputs. For reading, memories and register files can select one out of nine busses. For writing, memory and ALU outputs can select one of the six busses. Besides writing to one of the six global busses, the two memory outputs will always write to their own local bus. The two ALU outputs will write to a local bus if it is selected. The values one to six are reserved to write or read the busses one to six (Table 4.1). In normal operation mode, write enable of the

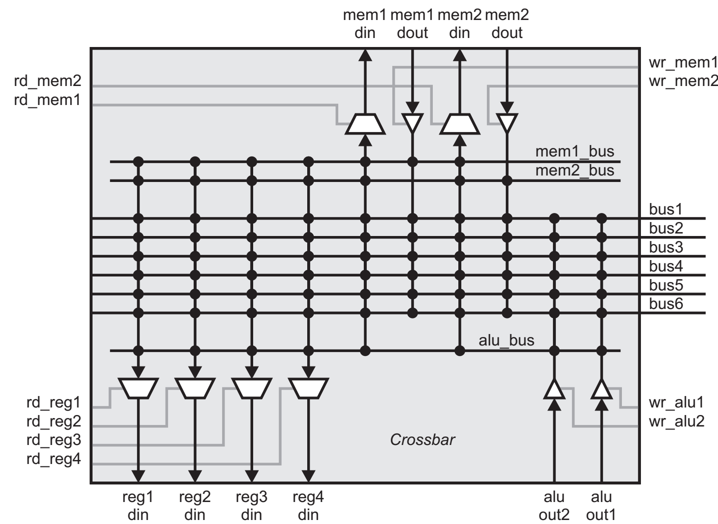

Figure 4.7: Part of the crossbar switch

Read select		Write enable	
value	source	value	destination
0000	no read from bus	000	no write to bus
0001	bus1	001	bus1
0010	bus2	010	bus2
0011	bus3	011	bus3
0100	bus4	100	bus4
0101	bus5	101	bus5
0110	bus6	110	bus6
0111	reserved	111	write to local bus
100-	mem1_bus		
101-	mem2_bus		
11--	alu_bus		

Table 4.1: Crossbar control signals

memory and registers will be automatically activated if the read address is not zero.

Three local busses are used in one part of the crossbar. Two local busses are used for the memory, because this way both memories can be read without using one of the six busses. (This is required to perform a 3D interpolation, Figure 3.6.) If two local busses were also used for the ALU outputs, then – during a twist – both (the ALU write and the memory read) addresses would have to change (Figure 4.8). Therefore only one local bus

is used for the ALU outputs, so only the crossbar signals controlling the ALU output have to change their value if two signals are being exchanged. A disadvantage of this implementation is that a global bus must be used if both ALU outputs are needed locally, which is energy inefficient.

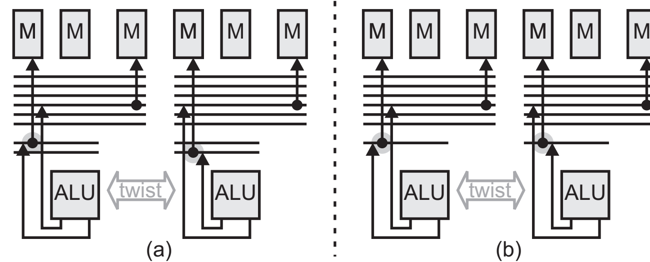

Figure 4.8: Local ALU bus (a) two busses (b) one bus

4.1.4 Memory and Counter

For storage, not only the register files, but also RAMs are used (Figure 4.9). They can be used to store intermediate values during calculations and tables to handle difficult functions. The memories are called 'local memories', because they are inside a processing part and the distance to an ALU is short. Each memory has its own counter, because the memories must be able to operate independently (e.g. to execute the FFT algorithm, they must operate independently).

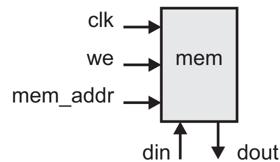

Figure 4.9: Memory

The counter (Figure 4.10) is used to address the memory. On the rising edge of the enable signal, *cnt_in* is automatically loaded into the counter as an offset value (Figure 4.11). If the enable signal stays on a HIGH value, the counter will increment ($dir = 0$) or decrement ($dir = 1$). If the counter is disabled (en=LOW), the counter will stop and stay at the same value. The signal *reset* will load zero as an offset value.

The FFT requires almost random memory access in a small range. Therefore, an extra input is added (reg_in) which allows eight random accesses of the three least significant bits. The least significant bits of the

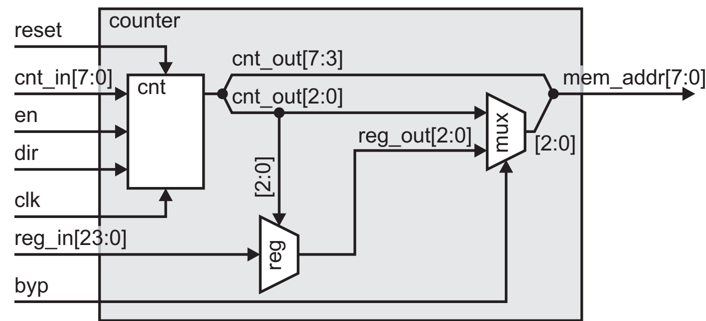

Figure 4.10: Counter

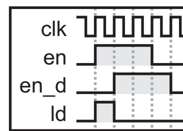

Figure 4.11: The internal load signal *ld* is automatically activated at the rising edge of the enable signal *en*

counter are used to select a part of *reg_in*. If *byp* is HIGH, the normal counting is bypassed and the random counting is selected.

Only when the chip is in normal operation mode the enable signal of the counter and the write enable of the memory can be HIGH. This is secured in hardware. If the chip is in system controlling mode (Section 4.2.4 on page 44), they will both be LOW. The write enable signal of the memory will automatically be HIGH if one of the busses of the crossbar is selected for reading (in normal operating mode) by *rd_memX* in Figure 4.7 (page 36).

To speed up the control of the counter and memory, and to make it easier to synchronize the memory with the register files, no register is used in the forward path of the counter (Figure 4.12). Therefore, memory locations can be accessed in the same number of clock cycles as registers (Figure 4.13), which makes the FPFA much easier to control.
Possible addressing modes are:

- FIFO. The signals for reset *(rst)* or load *(ld)* are used to implement a *first in first out* buffer.
- LIFO. The signal for direction *(dir)* is used to implement a *first in last out* buffer (stack).
- Restricted random access. Random access can be implemented in the small range of eight locations (3 bit).

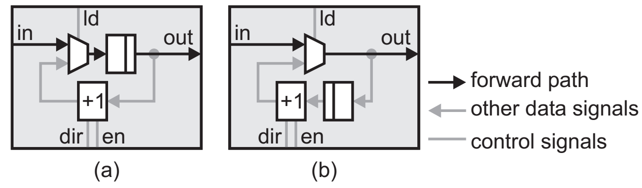

Figure 4.12: Counter (a) register in forward path (b) no register in forward path

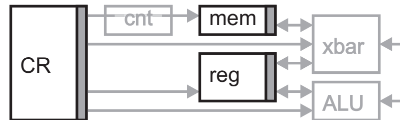

Figure 4.13: Memories can be controlled as fast as registers

4.2 Control

The ALU requires 33 control signals. However, each processing part requires not only control signals for the ALU (33), but also for registers (20), multiplexers and tristate drivers in the crossbar (36) and memory control (70). One processing part requires 159 control signals and each tile contains five parts. Therefore 795 control signals are needed for one FPFA tile. Of course, this number of signals is too large to be controlled directly by the tile controller. This problem can be solved by using configuration registers (CRs) and vertical microprogramming.

4.2.1 Configuration Registers

Mobile devices operate in a dynamically changing environment and must be able to adapt to the new environment. The key issue in the design of portable multi-media systems is to find a good balance between flexibility and high-processing power on one side, and area and energy-efficiency of the implementation on the other side. Reconfigurable systems have the potential to operate efficiently in these dynamic environments. The FPFA represents an efficient configurable computing solution, which is optimized for regular computationally-intensive applications, to reduce the control overhead.

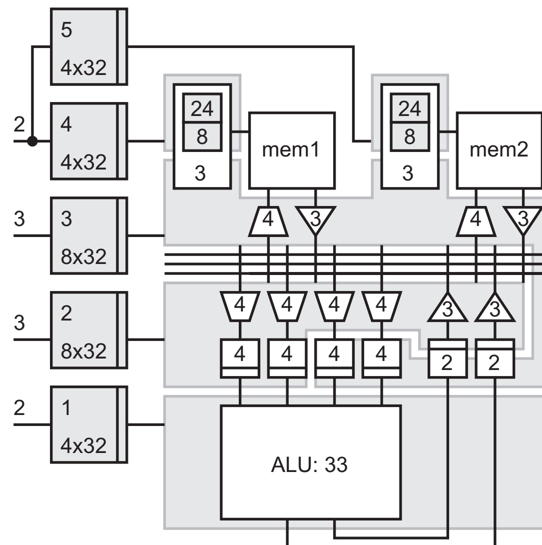

Figure 4.14: Configuration registers of one processing part

Configuration registers are used to store the context (i.e. the configuration control word) to select a particular datapath. The registers provide control bits for multiplexers, tristate drivers, registers and memories (Figure 4.14).

An FPFA tile contains five processing parts, each with five configuration registers. The configuration registers of the first processing part have the same functionality as the registers of other processing parts, but they do not have to contain the same data and they can be reconfigured independently.

In control mode, new contexts can be stored in the configuration registers; a new configuration can be loaded (Figure 4.15a). After filling the configuration registers, in normal operation mode, configurations can be switched; the configuration registers can be reconfigured by changing a few control signals (Figure 4.15b).

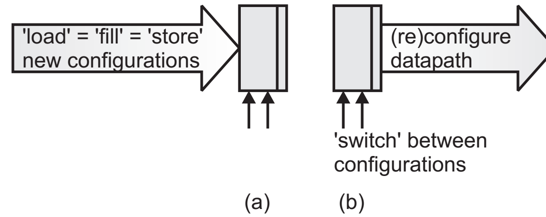

Figure 4.15: Two modes for the configuration registers: (a) in control mode new configurations can be loaded, (b) in normal operating mode configurations can be switched

To fill the configuration registers (in control mode), one of the them must be selected as destination. Therefore, all configuration registers have a number. The registers of the first processing part are numbered 1 until 5, the registers of the second processing part are numbered 6 until 10, and so on. Data can only be written into a configuration register if the registers number is selected.

A description of the signals controlled by the configuration registers of the first processing part (CR1 – CR5) is given in Appendix B and (as mentioned before) the functionality of other processing parts is the same. How the ALU, registers, crossbar, memory and counter are controlled by the configuration registers is discussed below.

Control of Datapath

Control of the datapath is very straight forward. Almost every signal is directly controlled by the configuration registers (Table 4.2 on page 42).

However, not every signal is directly controlled by the configuration registers. The addresses of the memories and the write-enable signals of register files and memories are controlled indirectly. Of course, the addresses of the memories are controlled by the counters. The write-enable signals of register files and memories are controlled by the crossbar signal selecting from which

Part	Description	Controlled by
ALU	Table A.2	CR1, CR3[23:22]
Reg. files	Subsection 4.1.2	CR2[7:0], CR3[21:14]
ALU output regs	Subsection 4.1.2	CR2[9:8], CR3[31:30]
Crossbar	Figure 4.7, Table 4.1	CR2[31:10], CR3[13:0]
Mem & Cnt	Subsection 4.1.4	CR3[29:24] CR4, CR5

Table 4.2: Datapath directly controlled by CRs

bus is read. The write-enable signal for register files is controlled indirectly by the crossbar signal *rd_reg** in CR2[31:16] and the memory is indirectly controlled by the crossbar signal *rd_mem** in CR3[7:0]. The write-enable signals can only be HIGH in normal operating mode (explained in more detail in Subsection 4.2.5). When writing to one of the configuration registers, automatically control mode is entered and write enable will become LOW.

4.2.2 Vertical Microprogramming

The configuration registers need to be controlled by 50 control signals. On one hand, this is a good reduction of the 795 control signals. On the other hand, with 50 bits one can make over a billion instructions and 60 instructions are enough for most DSP algorithms.

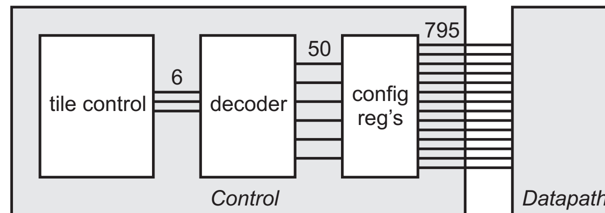

Figure 4.16: Vertical control in an FPFA tile: Tile control, decoder and configuration registers

In a horizontal microprogramming organization, each control signal corresponds to one bit in the micro word. Vertical microprogramming gives each unique set of control signals a separate code. This microprogramming procedure requires a decoder, which translates instruction codes into control

signals (Figure 4.16). This decoding can be done in an orderly fashion with the aid of storage. The code selects a word, and each bit of this word individually controls a signal.

Two-level control arises, when one instruction is decoded into control signals. This procedure is advantageous when a large number of signals is required for a complex datapath but only a relatively small number of different combinations occurs [Bla76]. Therefore a decoder is used to decode 6-bit instructions ($2^6 = 64$ different instructions) to 50 control signals.

4.2.3 Decoder

The decoder controls the configuration registers during normal operating mode. The decode unit has a data-input signal (di[31:0]), an address (decin[5:0]) and an output signal (decout[51:0]) (Figure 4.17). The data-input signal *di* is 32-bits wide, because the incoming data bus on the top entity of the FPFA is 32-bits wide (Subsection 4.2.5). The signal *decout* is 52-bits wide, and not 50-bits, because not only the configuration registers but also two tristate drivers for off-chip communication are controlled by the decoder. Two write-enable signals are used to write into the decoder, because the number of bits of the input signal is smaller than the number of bits of the output signal. One of them enables to write to the 32 least significant bits of *decout* and the other enables to write to the other bits; thus filling one location for the decoder output will take two clock cycles.

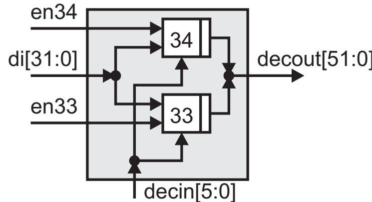

Figure 4.17: Decoder

Like the registers, the decoder has an address number that must be selected to write data into the decoder. Selecting the number will automatically enable the signals *en33* (least significant part) and *en34* (most significant part). To control the configuration registers, *decout* is connected to the address input of the registers (Figure 4.18). The first ten bits *(decout(9:0))* are connected to the configuration registers of the first processing part, the second ten bits to the second processing part, and so on. Of those ten bits, two bits are used to control CR1, three bits for CR2, three for CR3, and two for CR4 and CR5 together. Bit 50 of *decout* is used to enable a tristate driver between the data input *di* and the crossbar *bus1* and bit 51 is used

to enable a tristate driver between crossbar *bus2* and data output *do* (I/O signals for off-chip communication).

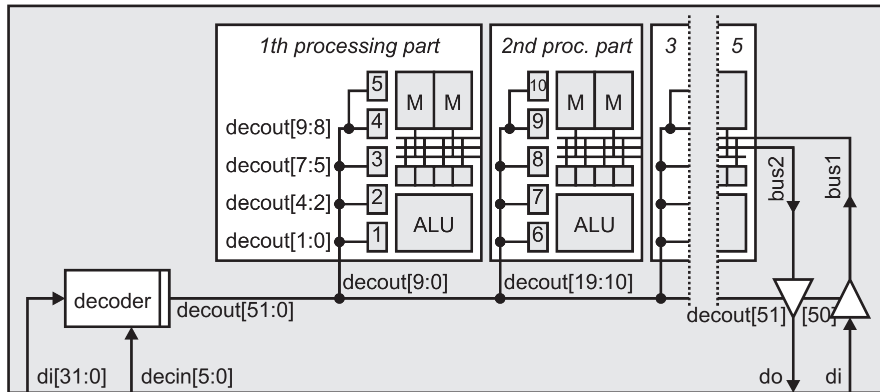

Figure 4.18: Decoder and configuration registers

4.2.4 Tile Control

In normal operating mode, *Tile control* is used to *switch* the configurations in the configuration registers (Figure 4.15). The system controlling mode is used to store new data into the configuration registers, this can be done internally and externally (Figure 4.19). Internally when Tile control runs a program from its *instruction memory* and externally when the FPFA is controlled by its I/O signals. A configuration can be loaded internally from the *data memory* when more different configurations are needed than the configuration registers can contain. Filling of the instruction memory, data memory and decoder cannot be done internally; it must be done externally (Table 4.3).

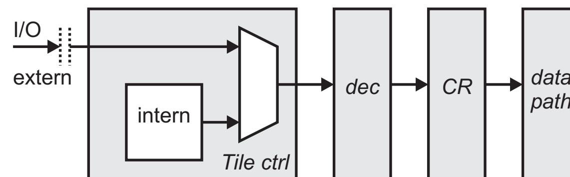

Figure 4.19: Internal and external control

This subsection will describe Tile control and all its components. In Tile control (Figure 4.20), two separations can be made; between data and instructions, and between internal and external control. This figure shows

	norm. op. mode	ctrl. mode
int. ctrl	calculate FFT, switch configurations	fill CR
ext. ctrl	fill local memories	fill CR, decoder, data or instr. RAM

Table 4.3: Some example FPFA activities for normal operating and control mode, and internal and external control

two multiplexers data_mux and instr_mux, that can switch between internal and external signals. In the next subsection (4.2.5, page 49), external control will be described. This section will describe internal control. Not only the internal instruction format, instruction-address calculation and a state diagram are discussed, but also how configuration registers, decoder, instruction memory and data memory are controlled by them.

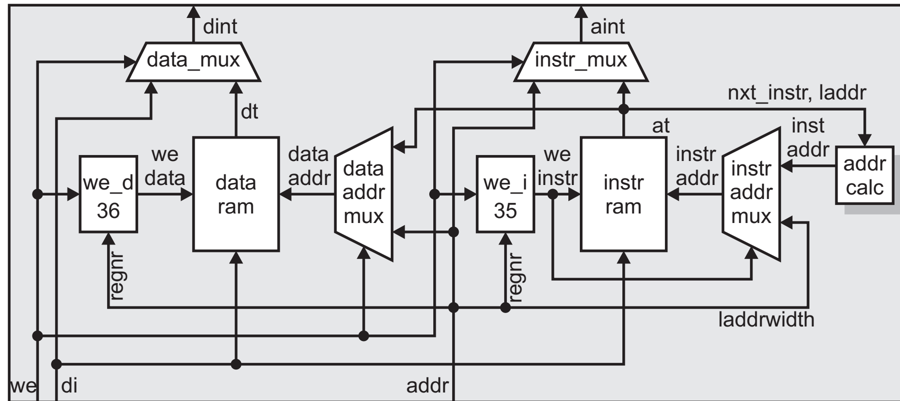

Figure 4.20: Tile control

Internal instruction format

Conventional processors work sequentially and thus are not suited to handle parallel data flow oriented applications. The main parts of the instruction format of those processors are the operation code and the address. The operation code selects which action must be performed – like data handling, logic operations or arithmetic operations – and the address locates the data with which this operation must be performed and where it must be stored.

Contrary to conventional processors, the FPFA is suited to handle parallel data flow oriented applications. To make this possible, three functions are required for the instructions for the FPFA.

1. *Switching between configurations.* Instructions must be able to control all the signals that control the datapath (indirectly through decoder and configuration registers) in normal operating mode.

2. *Loading of new configurations.* Instructions must be able to load new configurations into the configuration registers in control mode.

3. *Selection of the next instruction.* Instructions must be able to select the next instruction (increment, jump et cetera).

Therefore, the internal instruction *at* (in Figure 4.20) is divided in six parts (Table 4.4). The instruction memory is used to store the instructions, which can be executed.

Bit	Description
31..28	reserved
27..26	next address
25..23	number of waitstates
22..14	address for instruction or data memory
13.. 8	input (address) for decode unit *(decin)*
7.. 3	register number
2.. 0	configuration number

Table 4.4: Internal instruction

- The first two parts (configuration and register number) are used in control mode. The least significant eight bits [7:0] are used to write to a configuration register. Each register has a number, which can be selected by the bits [7:3] and the configuration can be controlled by the bits [2:0]. If a register contains only four configurations, instead of eight, bit [2] is not used. Control mode is (by definition) entered if the register number [7:3] is not equal to zero. The decoder, instruction memory and data memory cannot be filled internally, since numbers above 32 cannot be selected by five bits.

- The third part (input for decode unit) is used in normal operating mode. The bits [13:8] are used to control the decoder, and they indirectly control the configuration registers for switching between the configurations.

- The fourth part [22:14] is used to address the data or instruction memory. Normally in operating mode, the instruction address is just incremented and this part is not used. However, to make a jump, a destination address can be stored in this part of the instruction to

control the instruction memory. In the control mode, when data is written from the data memory into the configuration registers, this part is used to address the data memory.

- The fifth part [25:23] (number of waitstates) can be used for loops (nested loops are not allowed). If the value *"n"* of this number is not zero, address calculation will jump *"n"* times to the address in instruction part [22:14] and the last time this instruction is executed the address is incremented by one. Hence, every instruction in the loop is executed *"n+1"* times! Loops will be explained in more detail later.

- The last part [27:26] controls what the next address will be. Dependent of the value of this part of the instruction, the address calculation unit will reset, stop, increment or jump (Table 4.5). If the value of the fifth part [25:23] is not equal to zero (to implement a loop), this part [27:26] will be ignored, because the unit will make *"n"* jumps and one increment. When leaving control mode and entering normal operating mode, the instruction address is automatically reset.

Value	Action	Description
00	instr_addr=0	Reset to zero
01	instr_addr=instr_addr	Stop
10	instr_addr++	Increment by one
11	instr_addr=at[22:14]	Jump to address in instr. part [22:14]

Table 4.5: Next address (Instruction[27:26])

Instruction-Address Calculation

The control of the FPFA uses instructions that are executed in a specified order. Normally, instructions are executed in the numerical order of their addresses. However, the instruction address calculation unit must not only increment during the execution of the instructions, but also reset, stop or jump (as described above, in the fifth and sixth part of the instruction). We will now zoom in on the instruction-address calculation part of the Tile control to see how it is implemented (Figure 4.21).

The multiplexer inside the address calculation unit which generates *inst_add* is controlled by *nxt_add* (see mux in Fig. 4.21). The functionality of this multiplexer is described in Table 4.5. Normally – without a loop – *nxt_add* is equal to instruction part [27:26]. However, when the number of waitstates is not equal to zero but equal to the value *"n"*, this instruction part is ignored. The signal *nxt_add* will be *"n"* times "11" (jump) and the

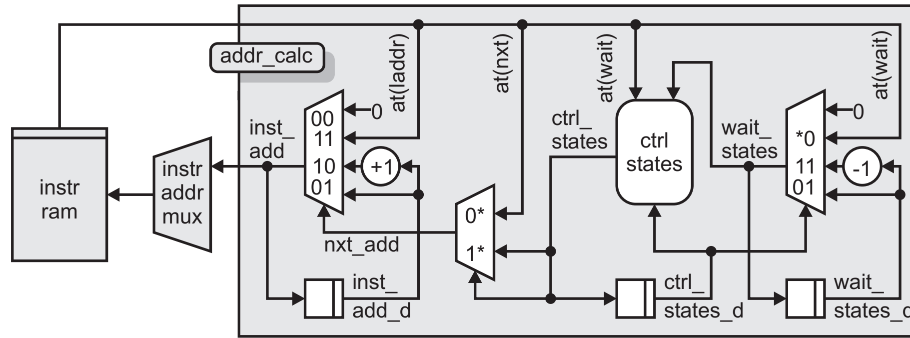

Figure 4.21: Instruction-address calculation

last time "10" (increment by one). The number of waitstates is decremented every time the instruction (which number of waitstates was not equal to zero) is executed.

State Diagram for Control

We will now zoom in on the control_states part of the instruction-address calculation unit. In Figure 4.22 each ellipse identifies a state of the control. The names of the states and the numbers are printed in the ellipse. The arrows indicate the transitions between the various states. The conditions for these transitions are shown next to the arrows. The next state is determined by the current state and the conditions. The states determine the actions of the program. In the states "10" and "11" the signal *nxt_add* is equal to the state value (i.e. "10" and "11" resp.), in the other states the signal is equal to instruction part *at[27:26]* (see mux controlled by *ctrl_states* in Fig. 4.21).

This state diagram is used to implement loops. In the state diagram we can see two important conditions:

- *at[wait]* the number of waitstates in instruction part [25:23]
- *waitst* the number of waitstates in a local counter

Normally – without a loop – control is in State "00" executing normal instructions. If the number of waitstates in the instruction is not zero, the loop is entered (State "11") and the number of waitstates in the instruction is assigned to the local counter. In the loop, State "01" is used when the number of waitstates in the instruction (at[wait]) is zero and State "11" (jump) when it is not. Every time the instruction with at[wait] non-equal to zero is passed, the local counter (waitst) is decremented by one. If the

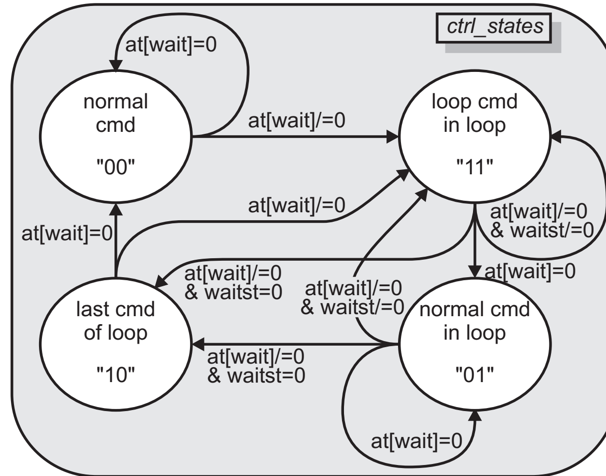

Figure 4.22: State diagram for ctrl_states

waitst is zero, State "10" (increment) is entered; so the unit will make a number of jumps and one increment. Dependent on the next instruction a new loop is entered or State "00" is entered.

4.2.5 External Control

The FPFA can be controlled externally by using the I/O signals of the chip (Figure 4.23, Table 4.6). The external control is used when the FPFA has to perform new tasks. If signal *wr* is HIGH, data can be written into the FPFA.

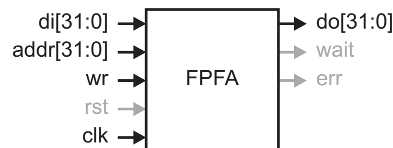

Figure 4.23: FPFA I/O

An address is used to select the destination of the data. The address selects the tile, the register and the configuration. If data is written into the instruction or data memories of the internal control, the memory address is selected (Table 4.7). Hence, there is much similarity between the external address and an (internal) instruction (Table 4.4). In normal execution mode, the FPFA is controlled by the instructions. However, if the external write *wr* is HIGH *and* the number of a tile is selected in the external address,

then the signal *we* (in Figure 4.20 on page 45) will also be HIGH. Data and instruction multiplexers will switch from internal signals to external signals. If the correct number is selected by address[8:3], data can be written to configuration registers (1 - 25), decoder (33, 34), instruction memory (35) and data memory (36). Writing to or reading from local memories or register files, can be done in normal operating mode; without selecting a number. In this mode, the decoder output *decout[51:50]* and the crossbar can be used to access the storage. How the I/O signals can be used and how they affect the FPFA will be illustrated in a few examples.

Signal	Type	Description
din[31..0]	I	data in
do[31..0]	O	data out
addr[31..0]	I	address
wr	I	write
clk	I	clock
rst	I	reset[1]
err	O	bus error[1]
wait	O	wait, not ready[1]

Table 4.6: FPFA I/O signals

Bit	Description
31..29	nothing
28..24	tile number
23..15	address for instruction and data memory
14.. 9	input for decode unit
8.. 3	register number
2.. 0	configuration number

Table 4.7: External address

Example 1: External Write to Control Parts

To write to a control part (configuration register, decoder, data or instruction memory), *wr* must be HIGH, *addr* must address the control part and *di* will have to contain the data. Four writes are performed in Figure 4.24, first a single write, then a burst of three writes. The single write, stores

[1]Not fully implemented

data into the decoder. The address "0x05000108" will write to tile 5, least-significant part of the decoder (number 33) and on address 00 of the decoder (as explained in Table 4.7).

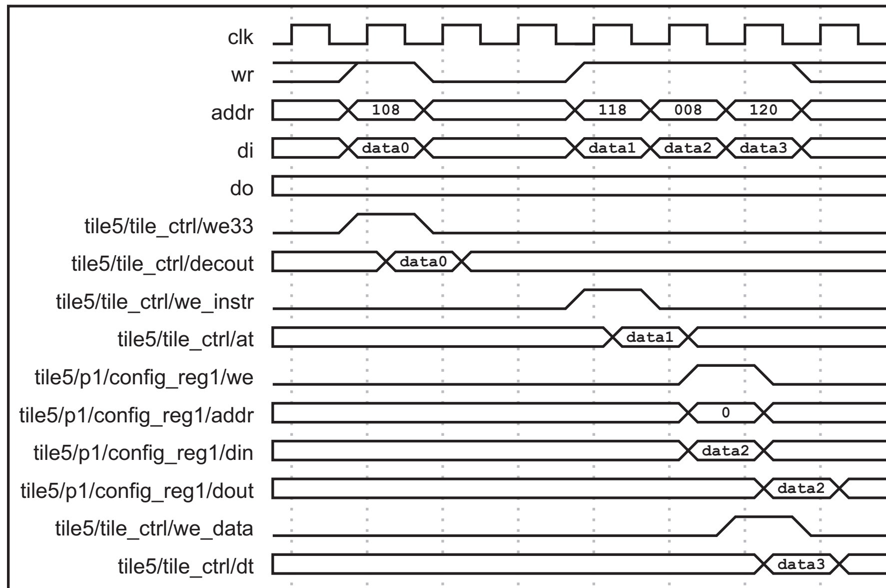

Figure 4.24: External write to decoder, instruction memory, configuration register and data memory

Instead of a single write to the control parts, also a burst of many writes can be performed. For example, in the same figure (4.24) we can see a sequence of three writes, to instruction memory, configuration register and data memory. One difference between the configuration register and the other control parts is the latency. It takes an extra clock cycle before data is written in the configuration registers. However, this does not mean that it takes an extra clock cycle to write during a burst. The address "0x05000118" will write to the instruction memory (number 35), address "0x05000008" will write to the first control register (number 1) and "0x05000120" will write to the data memory (number 36).

As we can see, making an external write to a control part can be done by simply selecting the number of this part in the address.

Example 2: External Write to Local Memories

Local memories cannot be written to by simply selecting a number in the external address. In the previous example control mode was entered by selecting a register number *addr[8:3]* to address a control part. Writing to the

local memories can only be done in normal operating mode, without selecting a register number, after configuring the configuration registers and the decoder. In this example, the decoder is controlled externally by *addr[14:9]*. How should the decoder and configuration registers (CRs) be configured? First, in control mode, new configurations must be loaded into the configuration registers and into the decoder. The tristate driver between *di* and *bus1* will have to be enabled by the decoder *(decout(50)='1')*, a memory has to read from *bus1* *(CR3(3:0)="0001")* and the counter should be enabled to count upward *(CR3(26:24)="100")*. Second, in normal operating mode, this configuration should be selected. The decoder can be controlled directly by the I/O signals and the output signal of the decoder *decout(7:5)* can be used to select a configuration of CR3 of the first processing part.

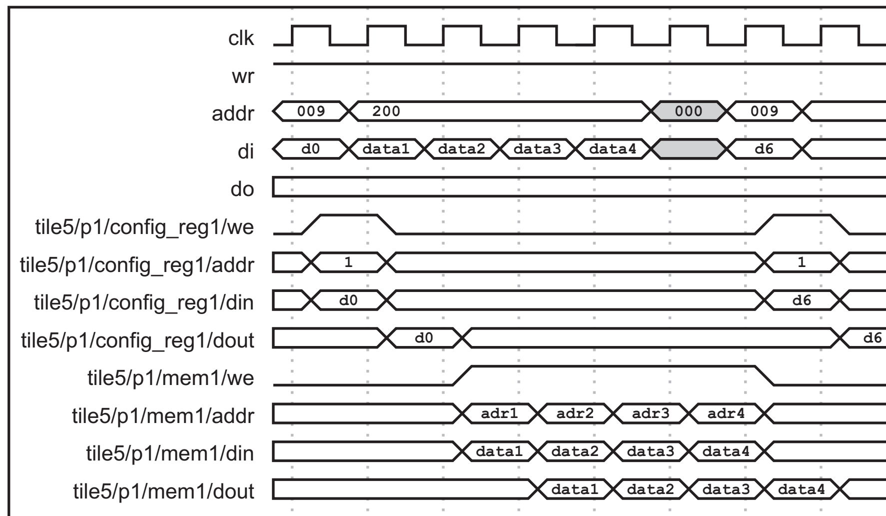

Figure 4.25: External write to local memories

Loading new configurations into the decoder and into the configuration registers should be done as explained in Example 1. Figure 4.25 emphasizes the second part in of a write to a local memory. This figure shows a write to configuration register *CR1*, then four writes to Memory 1 and after that another write to the register. The address "0x05000200" will select address "000001" of the decoder and the decoder will select the correct configuration of the configuration registers. In this case, it means data can be written to Memory 1 of the first processing part. To avoid that the tile will enter control mode before *data4* is written into Memory 1, one should not write to one of the control parts of this tile immediately, but (for example) insert an address "0x00000000" (like in the gray part of the figure).

So, writing data to the local memories or to the register files is more complex than writing to a control part. First, a configuration must be stored

in the decoder and the configuration registers, and then this configuration can be selected.

Example 3: Internal Read from Local Memories

Reading from the local memories can be done like described in Example 2, controlling the decoder with the external address *addr*. On the other hand, it can be done internally with the Tile control. For example, if *addr* would stop writing to control parts and if the instruction on address "0x000" would be a jump to address "0x021", and the instruction on address "0x021" would read Memory 1 for eight times, then the first output would appear after the sixth cycle on data output *do* (Figure 4.26).

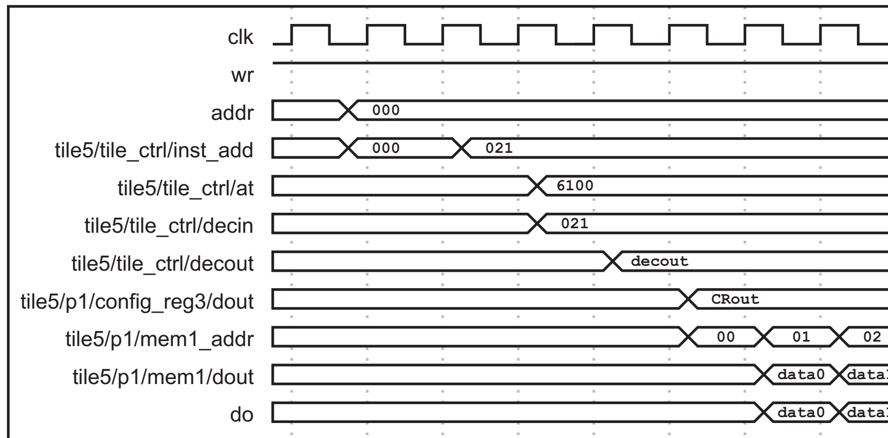

Figure 4.26: Read from memories using internal control

4.3 Evaluation

The goal of this thesis is to gather insight in the FPFA to reveal important benefits and pitfalls of the architecture. Therefore not only the design is described, but also a critical evaluation is given to optimize the design in the future.

- *ALU.* A fixed-point architecture was added to the implementation of the ALU datapath, to calculate fractions. Possible other improvements are the implementation of more logical operations in the f-functions (like shifting), and the implementation of the sign-magnitude representation to avoid that many bits change their value when going from a small positive to a small negative number.

- *Register files.* Instead of register files, a FIFO register could be used, to decrease the number of changes in the control signals, but register files are very flexible and suitable for low-power applications.

- *Output Register.* The output registers were added to avoid collisions or delays when both outputs of the ALU want to write to the same destination.

- *Crossbar Switch.* The crossbar was implemented using six global busses and three local busses for each processing part. The two memories can always write to their own local bus. The two ALU outputs share their local bus, because less addresses will have to change when they are exchanged. If the FPFA is programmed incorrectly, bus collisions can occur. Collisions can be prevented by using extra hardware (not implemented).

- *Memory.* The depth of the memory is 256, as advised by [Ash00].

- *Counter.* The counter can control the memory as a FIFO, LIFO and for random access in a small range. Although, this functionality was required by the mentioned algorithms, it might not be good enough for every other DSP algorithm. Therefore, it is recommended to look at the required functionality for other algorithms.

  The counter automatically loads an offset on the rising edge of the enable signal. An advantage of this automatic load is that no extra configurations are needed to load the offset first, and then start counting. A disadvantage can occur if the LIFO (stack) is implemented, and increment, decrement and stop are alternated frequently.

- *Configuration registers.* The datapath is controlled by so many signals that configuration registers are essential for a quick reconfiguration of the chip. However, the width of 32 is a trade off between flexibility, number of needed control signals, the depth (size) of the register and the time needed to fill all registers. It might be necessary to look very critically at the size of these registers and at the grouping of the control signals to simplify programming of the FPFA. To show how difficult the implementation of the configuration registers is, an example is given.

  This example illustrates the execution of a radix-2 FFT. First, the memory is enabled and the address starts to increment (T=0). At time T=1, data 'a' and 'b' appear on the output of the memory. At time T=2, this data appears on the output of the register files, so the ALU can start to calculate. The output of the ALU can be saved in a memory. Because both outputs cannot write to the same memory at

T	MemAddr	MemDo	Reg	ALU	MemDi	MemDi
0	0	–	–	–	–	–
1	1	a,b	–	–	–	–
2	2	a,b	a,b	FFT	out1	–
3	3	a,b	a,b	FFT	out2	out1
4	4	a,b	a,b	FFT	out1	out2
5	5	a,b	a,b	FFT	out2	out1
6	6	a,b	a,b	FFT	out1	out2
7	7	a,b	a,b	FFT	out2	out1
8	–	a,b	a,b	FFT	out1	out2
9	–	–	a,b	FFT	out2	out1
10	–	–	–	–	–	out2

Table 4.8: Radix-2 FFT example

the same time, one output is delayed and therefore the second memory receives its first data at time T=3.

On one hand we want to group the signals, therefore all control signals for the counters are in one configuration register. But on the other hand we do not want to group the signals because the number of needed configurations will grow exponentially. One memory starts to write (first configuration), then the other starts to read (second configuration), then another memory starts to read (third configuration). The FFT does not just write from Memory 1 to Memory 2, but also from Memory 2 to Memory 1, which doubles the number of configurations (six configurations). A configuration for the counters at rest is needed (seven configurations). If two local ALU busses would have been available, instead of one, the number of configurations would have doubled again.

The size of configuration registers is $width * depth$. The total width of all configuration registers must be 800, all signals must be controlled. To minimize the size of registers, we must minimize the depth, because the total width (800) is constant. On one hand the size must be reduced (by using configuration registers with a small width and on the other hand the number of control signals must be reduced (by using configuration registers with a large width). Configuration registers with a large width need more configurations and will therefore be larger. Configuration registers with a small width will need a smaller depth (less area), but there will be many to control. There is a trade off between size, flexibility and the number of control signals.

- *Control.* Each tile is controlled by a *Tile control.* One of the reasons to

make one control for each tile instead of a control for each processing part was the size. Another reason was that this is the most simple way to control the five processing parts, to make them work on one algorithm at the same time. Tile control is a small programmable controller, which generates a short instruction which is decoded by the (programmable) decoder, the decoder controls the configuration registers and finally the datapath is controlled by the configuration registers.

Conditional instructions are not implemented, but needed in many algorithms (e.g. Viterbi). A disadvantage of one control for five processing parts is that – especially when conditional logic is implemented – each processing part is less flexible.

- *Connections between tiles.* Interconnect between tiles is not implemented yet.
- *External communication.* For a low-power chip, the off-chip communication should be decreased to a minimum [Ste99]. This is done with internal RAM and registers. An address, write-enable and a data-input signal control the chip and an data output signal is used to present the results.

4.4 Conclusion

In this chapter, the implementation of datapath and control of the FPFA has been described, and an evaluation was given. Control is designed to reconfigure an FPFA tile. The design of the datapath is improved to execute the algorithms more efficiently. Both, datapath and control, are implemented in synthesizable VHDL. Logic synthesis is performed with FPGA Express, as discussed in the next chapter.

Chapter 5

Synthesis

The FPFA architecture is specified in VHDL and a demonstrator will be built on a Xilinx Virtex FPGA to verify the FPFA concept. Therefore synthesis is done with FPGA Express. FPGA Express (Synopsys) is a complete FPGA logic synthesis and optimization tool. With this tool optimized FPGA netlists can be created. The Project manager (Xilinx) is an application that integrates Foundation Series tools and FPGA Express into a unified environment.

This chapter is about the Virtex and the tools, how the tools can be used and what the results are.

5.1 Foundation Series

Foundation Series groups all related files into separate logical units called projects. VHDL designs are defined as elements in the project. The associated libraries as well as netlists, bitstream files, reports, and configuration files are all part of the project.

This section is divided in two parts. The first is about the Project manager and the second is about the tools that are managed by the Project manager.

5.1.1 Project Manager

The Project manager is an application that manages and supervises all Foundation Series tools involved in the design process. This environment includes tools such as HDL editor, logic simulator, flowchart for mapping, place-and-route and external third-party programs (like FPGA Express).

The Project-manager screen contains three sections (Figure 5.1).

- On the left side is the *Hierarchy Browser* consisting of a hierarchy tree of the project files on the *Files* tab and of the project versions on the

Versions tab.

- The upper right area includes the *Flow* tab showing the project flowchart. This area also contains the *Contents* and *Reports* tabs. From the Contents tab, one can view information on the Files and Versions shown in the Hierarchy Browser area. The system-created reports can be accessed from the Reports tab.

- The bottom *Console* tab displays errors, warnings, and messages. The *HDL Errors, HDL Warnings,* and *HDL Messages* tabs display information about synthesis results when a specific version of the project is selected.

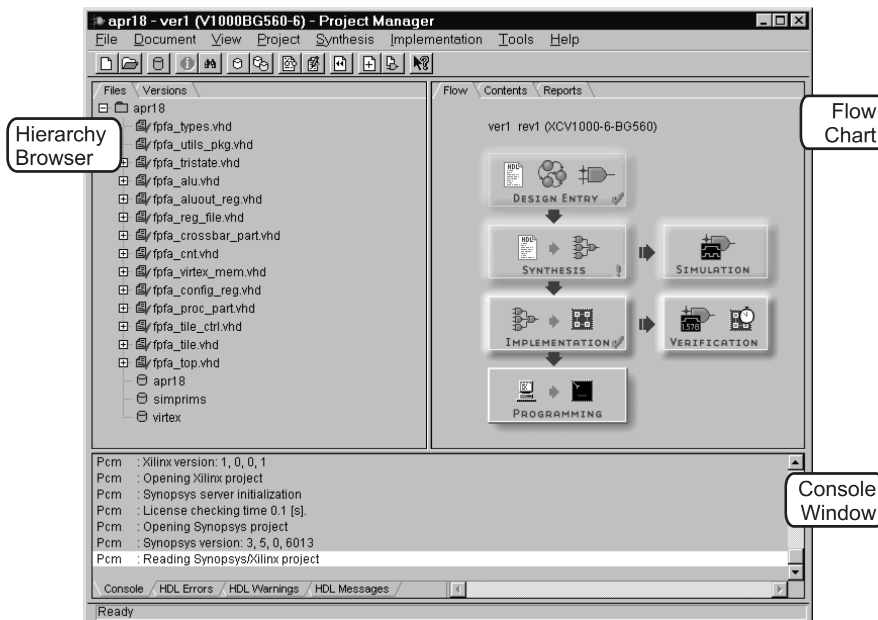

Figure 5.1: Project-manager window

5.1.2 Tools

HDL Editor

The HDL editor, a text editor, is designed to edit HDL source files created in VHDL. The editor checks the syntax for this language.

Synthesis

For design synthesis, FPGA Express from Synopsys is used. The synthesis tool accepts FPGA designs described in VHDL. The tool elaborates source

code, combines it with netlists from schematic entry, and then optimizes the description into a netlist that is ready for place-and-route.

Simulation and Verification

Simulation and verification tools are available to determine whether the timing and functional requirements of the design have been met. One can access the logic simulator from the project flowchart (upper right area) by clicking the *Simulation* button or the *Verification* button. Instead of the Project-manager tools, ModelSim is used for functional and timing verification (Chapter 6).

Implementation

Once the design is completed, physical implementation of the design can be started by clicking the *Implementation* button on the project flowchart. All the steps needed to obtain the final results are invoked automatically by the flow engine (Figure 6.1, page 66).

The flow engine will perform five steps (Figure 5.2).

1. *Translate* from the EDIF to the NGD (native generic design) format.
2. *Map* the NGD format to configurable logic blocks (CLBs) and input-output blocks (IOBs).
3. *Place* CLBs and IOBs and *route* interconnect.
4. Analyze *timing* for simulation.
5. Generate a *configuration file* for programming the FPGA.

Programming

When the design meets your requirements, the last step in its processing is programming the target device.

5.2 Virtex FPGA

The Virtex FPGA family delivers a programmable logic solution that enhances design flexibility while reducing time-to-market. The architecture is optimized for place-and-route efficiency and routing flexibility.
Virtex devices feature a flexible, regular architecture that comprises an array of configurable logic blocks (CLBs) surrounded by programmable input/output blocks (IOBs), all interconnected by a hierarchy of routing resources. Virtex FPGAs are SRAM-based, and are customized by loading

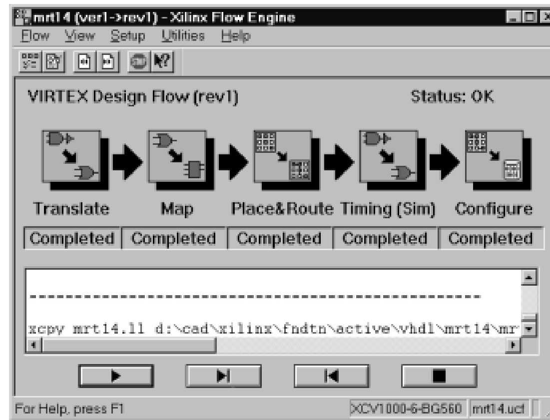

Figure 5.2: Flow engine

configuration data into internal memory cells. Designs can achieve synchronous system clock rates up to 200 MHz including I/O. Virtex inputs and outputs comply fully with PCI specifications, and interfaces can be implemented that operate at 33 MHz or 66 MHz.

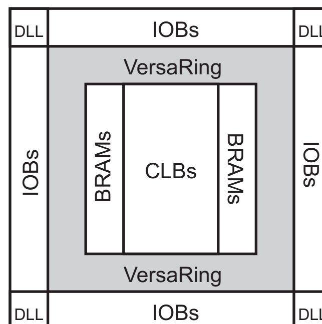

Figure 5.3: Virtex Architecture Overview

The Virtex, shown in Figure 5.3, comprises two major configurable elements: configurable logic blocks (CLBs) and input/output blocks (IOBs). CLBs provide the functional elements for constructing logic and IOBs provide the interface between the package pins and the CLBs. CLBs interconnect through a general routing matrix (GRM). The VersaRing I/O interface provides additional routing resources around the periphery of the device. This routing improves I/O routability and facilitates pin locking. The Virtex architecture also includes block memories of 4096 bits each, 3-State buffers (BUFTs) and items for clock-distribution delay compensation and clock do-

main control.

5.3 Logic synthesis

This section describes the basic steps to perform logic synthesis and to map, place and route the FPFA design with the Project manager onto a Virtex. FPGA Express tries to optimize the design and during implementation a VHDL netlist is generated.

1. **Invoke Project manager**
   - Project manager F3.1i and FPGA Express FE3.5 were used.
2. **New project**
   - Select: File > New Project...
   - Type the name of your project in the *name* field.
   - Type the path to your project in the *directory* field.
   - Select type: *F3.1i*.
   - Select flow: *HDL*.
3. **Add files to project**
   - Select: Project > Add Source File(s)...
   - Add files in the order as specified in Figure 5.1.
4. **Synthesis**
   - Select: Synthesis > Force Analysis of All Sources.
   - If analysis completed successfully, a green $\surd$ will appear on the the flow chart (Figure 5.1).
   - Push the *Synthesis* button on the flow chart or select: Synthesis > Synthesis.
   - Select: *fpfa_top* in as top level.
   - SET the synthesis settings:
     - Optimize for: *area*
     - Effort level: *low*
     - Target clock frequency: *1.*
   - Select family (of the target device): Virtex.
   - Select (target) device: V1000BG560.
   - Push *OK* and push the *Synthesis* button to start synthesis.
   - If synthesis completed successfully, a green $\surd$ will appear on the the flow chart.

5. **Synthesis**

   - Push the *Implementation* button on the flow chart or select: Implementation > Implement design.
   - Push the *Options* button and select *ModelSim VHDL* for simulation.
   - Push *OK* and push the *Implementation* button to start implementation. The flow engine will pop up.
   - If implementation completed successfully, a green $\surd$ will appear on the the flow chart.

5.4 Results

When implementing the FPFA on an FPGA, the goal was not to make a high-speed low-power application. The first goal was to make the whole design of one tile fit on an FPGA. It appeared that five processing parts, with just a few small configuration registers and without control, already took 77% of the SLICEs (Table 5.1).

Number of External GCLKIOBs	1 out of 4	25%
Number of External IOBs	123 out of 404	30%
Number of BLOCKRAMs	20 out of 96	20%
Number of SLICEs	9464 out of 12288	77%
Number of GCLKs	1 out of 4	25%
Number of TBUFs	2240 out of 12544	17%

Table 5.1: Device utilization summary (datapath of one tile on XCV1000-BG560)

This first design helped to make an estimation of how large the size of the eventual design could be. Because this small design already took 77% of the SLICEs, the control part had to be very small. This was one of the reasons to choose one control part for five processing parts; i.e. Tile control. Another reason was that this is the easiest way to make the five ALUs work on one algorithm at the same time. The eventual design takes 96% of the SLICEs (Table 5.2).

Table 5.3 shows a maximum frequency of 6.5 MHz. Three recommendations can be given to increase the speed.

- The FPFA is mapped onto an FPGA, therefore the design is slower than it could be. Implementing the FPFA as ASIC can increase the speed.

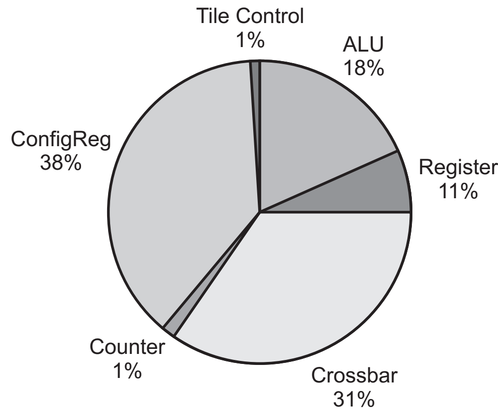

Figure 5.4: Utilization diagram (percentage of SLICEs of one tile)

Number of External GCLKIOBs	1 out of 4	25%
Number of External IOBs	84 out of 404	20%
Number of BLOCKRAMs	27 out of 32	84%
Number of SLICEs	11892 out of 12288	96%
Number of GCLKs	1 out of 4	25%
Number of TBUFs	2420 out of 12544	19%

Table 5.2: Device utilization summary (one tile on XCV1000-BG560)

Minimum input arrival time before clock	38.394 ns
Minimum output required time after clock	149.012 ns
Minimum period	154.041 ns
Maximum frequency	6.492 MHz

Table 5.3: Timing summary (one tile on XCV1000-BG560)

- Synthesis was not optimized for speed, but for area, to make the design fit on one Virtex. Optimizing for speed can increase the speed.
- The high period is caused by a very long path, coming from the register files of processing part 5, going through the east-west connections of all the parts to the crossbar. This longest path (through all the processing parts) is often not used. Not using this very long path can speedup the frequency by a factor three.

To make area and timing estimations for an ASIC design, synthesis was performed by Ambits synthesis tool *BuildGates*. Results are given in Figure 5.5 and Table 5.4.

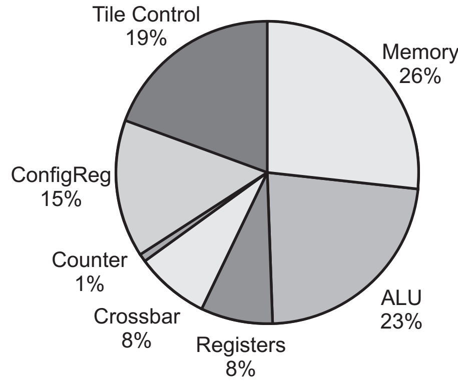

Figure 5.5: Area diagram (percentage of area of one tile), total area = 2.6 mm^2 in CMOS18

Arrival Time	42.785 ns
Minimum period	42.785 ns
Maximum frequency	23.373 MHz

Table 5.4: Timing summary (CMOS18)

The area of the chip without memories is 1437126.670 μm^2. Ten local memories are added (26% memory) and in Tile control, memories are used for the decoder, instructions and data (18% memory out of 19% control). The arithmetic and logic unit consumes almost a quarter of the area. Stekelenburg [Ste99] expected 5 mm^2 for the datapath in a 0.35 μm process. Therefore an area between 2 and 3 mm^2 for the whole tile (including configuration registers and Tile control) in CMOS18 was expected.

The maximum frequency in CMOS18 (23 MHz) is determined by the very long path through all five processing parts.

Quantitative comparisons to other architectures are difficult because of differences in technology, application details et cetera. For comparison, performance results of the RaPiD [Cro99] are cited. The RaPiD architecture, optimized for performance rather than power, consists of reconfigurable cells, containing one multiplier, three simple ALUs (without a multiplier), three memories and six registers, so it can be compared with one fifth of an FPFA tile. The longest path delay of a RaPiD cell is 8 ns (clock frequency >100 MHz) and the area of one cell is 2.25 mm^2 (0.35 μm process). So, the area appears to be approximately the same and the FPFA clock frequency appears to be low. Compared to GPP and FPGA-based architectures, the control overhead is substantially reduced. The overhead is less than 50% (ALU, registers and memories are functional units) instead of 98% [Cro99].

Chapter 6

Simulation

Design verification is an important aspect of each design project and it consists of functional simulation and timing simulation. Before synthesis, one should take the time to verify that the design actually does what it is intended to do. Therefore functional simulation must be done. Functional simulation can be done after the schematic has been entered; before a HDL file has been synthesized (Figure 6.1). Functional simulation gives information about the logic operation of the circuit. It does not provide any information about timing delays. Timing simulation will tell you how fast signals travel through the gates and how fast the overall circuit can be operated. In order to do a timing simulation, one needs to implement the design in a specific target device.

So first, functional simulation is done, using ModelSim, to verify the functionality of the design. Then FPGA Express and the Xilinx tools are used for synthesis, mapping and place-and-route to generate a VHDL netlist with a timing simulation model. After synthesis and implementation, ModelSim is used again to perform timing simulation. Synthesis and implementation were already explained in the previous chapter. In this chapter, functional simulation will be explained first, followed by timing simulation, test bench and results of simulation.

6.1 Functional Simulation

This section describes the basic steps to compile and simulate the RTL design. Functional simulation verifies the functionality of the design prior to synthesis.

1. **Invoke ModelSim**

   - ModelSim EE 5.4a was used.

2. **New Project**

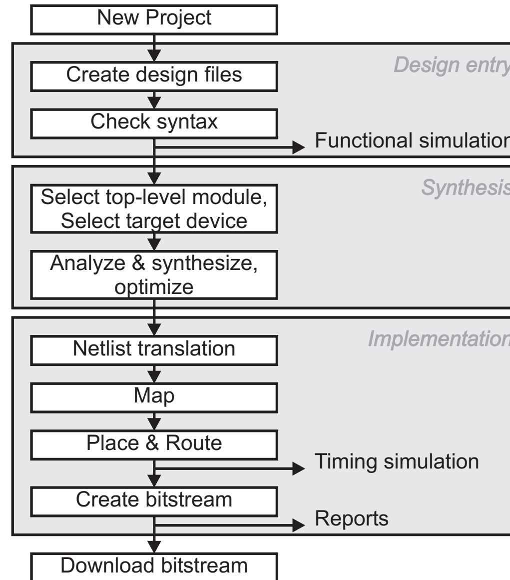

Figure 6.1: Design flow

- Select: File > New > New Project.
- Type the path to your project in the *home* field (without the name of the project).
- Type the name of your project in the *name* field.

3. **Simulation options**

   - Select: Options > Simulation > Assertations.
   - Select break on assertion: *Error*.
   - Select ignore assertions for: *Warning* and *Note*.

4. **Compile and simulate**

   - Select: Macro > Execute macro > fpfa_modelsim.do (Appendix C.1).

     This macro will execute:

     > `vcom fpfa_*.vhd`
     to compile the VHDL files.

     > `vsim -t ps work.fpfa_tb`
     to invoke the simulator.

     > `add wave -radix hexadecimal /fpfa_tb/*`
     to add signals to the wave window.

> `run -all`
> to run the simulation.

The Wave window shows the results of the simulation. In the Wave window, one can see the results as waveforms and their hexadecimal values (Figure 6.2). This figure shows a read from the local memories using internal control, like in Example 3 in Chapter 4 (Figure 4.26).

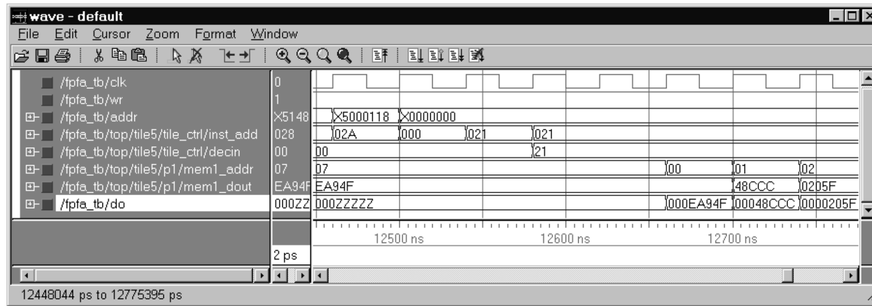

Figure 6.2: Functional simulation

6.2 Timing Simulation

This section describes the basic steps to compile and simulate the gate-level design. The timing of the design is verified after synthesis and place-and-route. The steps are like functional simulation (Section 6.1), first invoke ModelSim, make a new project, select the correct simulation options and finally execute a macro.

Before executing the macro for the first time, one has to compile the SIMPRIM library. The SIMPRIM libraries are used for timing simulations. The VHDL source code for the libraries can be found at: $XILINX/vhdl/src/simprims. This directory contains the following files.

- simprim_Vpackage.vhd (VITAL Table Package)
- simprim_VITAL.vhd (VITAL Architecture and Configurations)
- simprim_Vcomponents.vhd (VITAL Component Package)

The libraries are completely design independent. Therefore they are used for the timing simulation of any Xilinx device family and they only have to be compiled once, using the command `vcom`.

To verify the netlist, another macro has to be selected than with functional simulation and other files are compiled (Appendix C.2). This macro will execute:

> `vcom time_sim.vhd`
> to compile the VHDL netlist.

> `vcom fpfa_types.vhd fpfa_test_pkg.vhd fpfa_tb.vhd`
> to compile the VHDL files.

> `vsim -t ps -sdftyp /fpfa_tb/top=time_sim.sdf fpfa_tb`
> to invoke the simulator.

> `add wave -radix hexadecimal /fpfa_tb/*`
> to add signals to the wave window.

> `run -all`
> to run the simulation.

The Wave window in Figure 6.3 shows a read from the local memories, like Figure 6.2 does. However, in this figure the results of timing simulation are represented.

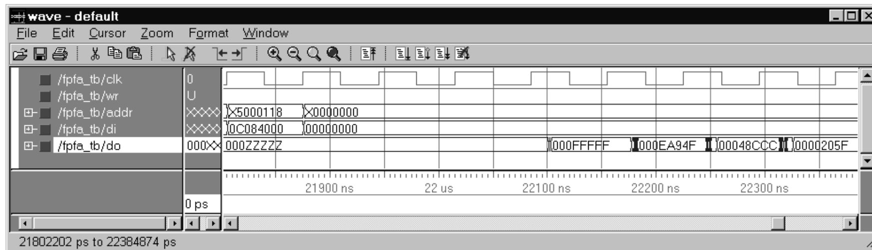

Figure 6.3: Timing simulation

6.3 Test Bench

Before synthesis, one should take the time to verify that the design actually does what it is intended to do. Therefore functional simulation must be done. To verify the proper operation of the circuit over time in response to input stimuli, a test bench had to be written. The test bench, described in VHDL, applies stimuli to the FPFA design and verifies that the simulated circuit does what it is intended to do. VHDLs file features (IN/OUT) are used to apply stimuli in the form of test vectors. In the input file (.cmd), commands like 'write', 'check' and 'run' are used. The output file (.out) is used to write the simulation results to a file for later analysis. This file contains all executed commands, with expected and actual outputs. A comprehensive, re-usable test bench was made, by separating the test bench (virtual socket for FPFA), a test package (command file reader), command

file and output file (Figure 6.4). Each of them is discussed in the following subsections (6.3.1 - 6.3.3).

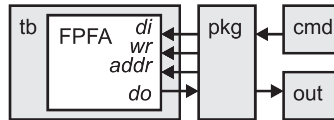

Figure 6.4: Test bench, test package, command file and output file

6.3.1 Test Bench

Both, design before and after synthesis, should be embedded in a virtual socket. We cannot use the same test bench before and after synthesis, because some redundant signals were eliminated. Therefore another test bench was made for the netlist after synthesis.

The test bench uses three processes that operate concurrently; the FPFA design, a clock generator and a test vector generator. The FPFA design is instantiated as a component in the test bench, the second process (clock) describes a background clock, while the third process (a command-file reader) applies the stimuli and checks the results. Actually the third process is just a function call, calling a function in the test package.

6.3.2 Test Package

The test package contains functions. The most important function is invoked by the test bench, its main task is to read the characters from the command file, return standard logic test vectors to the test bench and compare expected values with actual output values from the design. Besides the main function, other functions are used for conversions between decimal (dec), binary (bin) and hexadecimal (hex) strings (str), integers (int) and standard logic vectors (slv) (Figure 6.5).

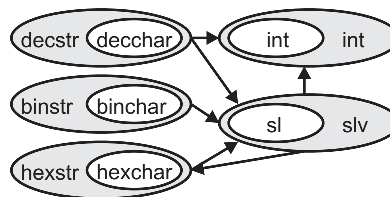

Figure 6.5: Conversion functions

Four commands can be read by the test package: `write`, `check`, `read` and `run`. An example of each command and a short description of the arguments is given in the Tables 6.1, 6.2 and 6.3.[1] The example is in the first row of the table and the arguments in the rest of the table.

- The `write` command (Table 6.1) writes data of the arguments 1 - 5 to address *addr* and the data of the last (6th) argument to the data-input signal *di* of the FPFA. The address arguments are encoded binary and the data argument is encoded hexadecimal, to simplify debugging. This command automatically makes the write-enable signal *wr* HIGH and runs simulation for one clock cycle.

write 00101 001001001 010101 001100 010 000F3E12			
Argument	Description	Size	Format
1	tile number	5	bin
2	address	9	bin
3	decin	6	bin
4	register number	6	bin
5	configuration	3	bin
6	data *di*	8	hex

Table 6.1: Write format, see Table 4.7

- The `check` command (Table 6.2) verifies whether the first argument is equal to the data output signal of the FPFA *do*. If the error is not larger than three, no warning is given (to exclude false alarms because of rounding). In fixed-point notation this means the error should be smaller than 10^{-4} (for DSP applications: not hearable and not visible). This command does not change *addr*, nor *wr*. Neither does it run simulation for one clock cycle, unless `run` is written as second argument. This way, write and check can be performed at the same time.

check 000F3E12 (or) check 000F24A7 run			
Argument	Description	Size	Format
1	data *do*	8	hex
2	(run)	0, 3	char

Table 6.2: Check format

- The `read` command makes the write-enable signal *wr* LOW, without running simulation for one clock cycle.

[1]No table for *read*, because *read* has no arguments

- The `run` command (Table 6.3) runs simulation for one clock cycle if no argument is used. Although, the *run* argument in the *check* command cannot have an argument, this *run* command can. If an argument is used, the decimal number in this argument determines the number of clock cycles that simulation will run. So, if the argument is 5, the simulation will run five clock cycles.

run (or) run 5 (or) run 36			
Argument	Description	Size	Format
1	(number)	0, 1, 2...	dec

Table 6.3: Run format

Besides these commands, the characters `#`, `//` and `--` can be used to start comment. The rest of the line, behind these characters, is interpreted as comment.

6.3.3 Command File

In the command file the commands (like mentioned in the previous subsection) are listed. The example given in this subsection shows a read from the local memories, just like Example 3 of Chapter 4 (Figures 4.26, 6.2 and 6.3).

Input

In the first line of the example below (which is the 296th line of a larger file), a write to the instruction memory is performed (`100011` is number 35, the number for this memory). To read from a local memory, the tile should be in normal operation mode, so the address should not select a control part of the tile. Therefore, the address becomes zero in the second line of the example. Another option was to select another tile or to execute a `read` command. It takes six cycles before the data is on the output and the command `write` already ran simulation for one clock cycle. Therefore `run 5` is in the third line. Then we can start checking the output. After each check, simulation can run for one clock cycle.

```
write 00101 000000000 000000 100011 000 0C084000 --goto 33
write 00000 000000000 000000 000000 000 00000000
run 5
-- (comment)
-- ...
-- Check:
check 00048CCC run
check 00002060 run
```

Output

In the output file, line numbering of the input file is added to simplify debugging. The line numbering of the output file is not equal to the line numbering of the input file, because comments are removed (see line 298, 336).

```
296: write 00101 000000000 000000 100011 000 0C084000
297: write 00000 000000000 000000 000000 000 00000000
298: run 5 -> 5
336: check 00048CCC ?= 00048CCC -> OK; run
337: check 00002060 ?= 0000205F -> OK; run
```

Hence, although "0x00002060" is not equal to 0000205F, simulation says "OK". As mentioned before, warnings for small rounding differences are omitted. When the differences are larger than three (3), a warning is given. For example:

```
337: check 00002060 ?= 000XXXXX -> WARNING 8288; run
```

6.4 Results

Writing to configuration registers, decoder, data and instruction memory, writing to local memories and reading from local memories, using ALU, register files, output registers, crossbar and local memories, these operations and the components have been verified with functional and timing simulation. However, the design is so large, that it has become impractical to test the circuit exhaustively; writing all test vectors for every combination is impractical, or perhaps even impossible.

One of the implemented algorithms is the FFT. This algorithm was chosen because it requires the highest flexibility of the three algorithms discussed in Chapter 3. Data was written into the local memories, configurations were loaded, calculations were performed and data on the output signal was checked. Functional and timing simulations (with a 15 MHz clock) were performed. All output values were within the 10^{-4} range, which gives us confidence in the functional correctness of the design.

Chapter 7

Conclusions and Recommendations

7.1 Conclusions

Looking at the trade off between flexibility, speed and energy consumption, an FPFA seems the logical step ahead, providing a new and better way for the implementation of highly repetitive, computationally-intensive DSP algorithms. It has a reconfigurable datapath to create flexibility and it has short design times, because execution of another application does not require the design of a new integrated circuit. Many computations can be performed in parallel and because the datapath has a low power radius, it is energy efficient.

Three often-used DSP algorithms have been studied to find the requirements for the functionality of the FPFA: linear interpolation, FIR and FFT. Of course, these are only a few of the algorithms that the FPFA should be able to handle. The three algorithms have been mapped onto the FPFA successfully.

Previous designs and optimizations of the FPFA datapath have been studied. The datapath has been improved, so DSP algorithms can be executed more efficiently, and control has been designed. The datapath of an FPFA tile contains five processing parts. Each processing part uses an ALU, four register files, a crossbar and two memories. Control consists of three parts: configuration registers, decoder and Tile control. Vertical programming is used because the rather complex datapath with many control signals requires a relatively small number of different combinations. The FPFA has been designed and implemented and the FPFA architecture is specified in a high-level description language (VHDL).

Logic synthesis has been performed and the design was mapped and place-and-routed onto the Virtex. This proves that the FPFA implementation fits on a Virtex and it is specified in *synthesizable* VHDL. One FPFA tile

appears to have an area of 2.6 mm^2 (CMOS18), as expected, and it can run at 23 MHz in a CMOS18 process and 6,5 MHz on a Virtex. A VHDL netlist was created by the synthesis tools to perform not only functional, but also timing simulation. A test bench has been created to perform simulations and a realistic application has been simulated. The implementation of an algorithm has shown that programming an FPFA by hand is not trivial. The FFT algorithm has been simulated successfully. Simulations proved that the FPFA implementation is functionally correct, before and after synthesis.

7.2 Recommendations

The design was already evaluated in Section 4.3, many recommendations were already given. The most important recommendations are listed below.

- Investigate what logical operations are needed for the f-functions of the ALU (shifting).
- Implement the sign-magnitude representation.
- Investigate whether the automatic load should be eliminated and whether more functionality is required for the counters.
- Implement conditional instructions.
- Implement interconnect between tiles.
- Further research is required to investigate what a good size is for the configuration registers and how the control signals should be grouped.
- Implement low-power recommendations of other theses (latches).
- Optimize area of the FPFA by optimizing one processing part by hand.
- Optimize speed of the FPFA (pipelines).
- Design a tool, to avoid mapping and implementation of a DSP algorithm by hand.

Bibliography

[Ash00] A. Ashtari. *Design and optimization of an FPFA*. M.Sc. Thesis Report code: EL-S&S-040N99, University of Twente, Enschede, The Netherlands, January 2000.

[Ber96] A.J.W.M. ten Berg et al. *Computer organisatie*. Syllabus Course code: 213110, University of Twente, Enschede, The Netherlands, October 1996.

[Bla76] G.A. Blaauw. *Digital System Implementation*. ISBN: 0132122413, Prentice-Hall Inc, United States of America, 1976.

[Büc98] M. Büchli. *Optimalizatie van een Field Programmable Function Array*. Technical Report Report code: EL-S&S-065N98, University of Twente, Enschede, The Netherlands, November 1998.

[Cha99] Chameleon. *Chameleon homepage*. University of Twente, http://chameleon.ctit.utwente.nl, The Netherlands, 1999.

[Cha00] A. Chass et al. *Efficient software implementation of the Max-log-MAP Turbo decoder on the StarCore SC140 DSP*. DSP conference proceedings, Motorola Semiconductor, Herzelia, Israel, 2000.

[Cro99] D.C. Cronquist et al. *Architecture design of reconfigurable pipelined datapaths*. Paper for 20th ann. conf. on adv. research in VLSI 1999, University of Washington, Seattle, United States of America, 1999.

[Ebe97] C. Ebeling et al. *Configurable computing: the catalyst for high-performance architectures*. Article in ASAP 97, University of Washington, Seattle, United States of America, 1997.

[Hey00] P.M. Heysters et al. *Exploring Energy-Efficient Reconfigurable Architecture for DSP Algorithms*. Article in progress2000, University of Twente, Enschede, The Netherlands, October 2000.

[Jan99] P. Jansen. *Systeem programmering*. Syllabus. Course code: 211004, University of Twente, Enschede, The Netherlands, December 1999.

[Kla98] C.E. Klaasen. *A VHDL based implementation of an ALU of a Field Programmable Function Array.* M.Sc. Thesis Report code: EL-BSC-001N98, University of Twente, Enschede, The Netherlands, March 1998.

[Mar98] R. Marsman. *Optimalisatie van het FFT algoritme en verbetering van een FPFA.* Technical Report Report code: EL-S&S-064N98, University of Twente, Enschede, The Netherlands, November 1998.

[Pac00a] Pact Corporation. *XPP processing models; eXtreme Processing Platform.* Whitepaper, http://www.pactcorp.com, Munich, Germany, 2000.

[Pac00b] Pact Corporation. *XPP technology; eXtreme Processor Platform.* Datasheet, http://www.pactcorp.com, Munich, Germany, 2000.

[Pac00c] Pact Corporation. *XPU128; eXtreme Processing Unit.* Datasheet, http://www.pactcorp.com, Munich, Germany, 2000.

[Rem97] G.B. Remmelink. *Design of an architecture for a Field Programmable Function Array.* M.Sc. Thesis Report code: EL-BSC-038N97, University of Twente, Enschede, The Netherlands, June 1997.

[Rob87] R.A. Roberts. *Digital signal processing.* ISBN: 0201163500, Addison-Wesley, United States of America, 1987.

[Smi95] J. Smit and J.A. Huisken. *On the energy complexity of the FFT.* Article in Proceedings of PATMOS 95, pp.119-132 ISBN 3-8142-0526-X, University of Twente, Enschede, The Netherlands, 1995.

[Smi98] J. Smit and H. Snijders. *VLSI system design and VLSI Signal Processing.* Syllabus. Course codes: 121713 and 122733, University of Twente, Enschede, The Netherlands, December 1998.

[Smi00] G.J.M. Smit et al. *Mapping the SISO module of the Turbo decoder to a FPFA.* Article, University of Twente, Enschede, The Netherlands, 2000.

[Ste99] M. Stekeleburg. *Optimization of an FPFA.* M.Sc. Thesis Report code: EL-S&S-013N99, University of Twente, Enschede, The Netherlands, April 1999.

[Val96] M.C. Valenti. *An introduction to Turbo codes.* Technical report, West Virginia University, Morgantown, United States of America, May 1996.

Appendix A

Arithmetic and Logic Unit

Signal	Size	Type	Description
a	w	I	data input
b	w	I	data input
c	w	I	data input
d	w	I	data input
east	2*w	I	double precision data input
west	2*w	O	double precision data output
out1	w	O	data output
out2	w	O	data output

Table A.1: ALU data signals, see Figure 2.2

Signal	Size	Type	Description
ctf3(5)	1	I	’0’ : op1:=in1,
			’1’ : op1:=–in1.
ctf3(4)	1	I	’0’ : op2:=in2,
			’1’ : op2:=–in2.
ctf3(3:1)	3	I	”000”: f:=0,
			”001”: f:=op1,
			”010”: f:=op2,
			”011”: f:=op1+op2,
			”100”: f:=min(op1,op2),
			”101”: f:=max(op1,op2)
ctf3(0)	1	I	’0’ : out:=f
			’1’ : out:=abs(f)
ctf2	6	I	see ctf3(5:0)
ctf1	6	I	see ctf3(5:0)
selmx	2	I	”00” : mx:=a
			”01” : mx:=b
			”10” : mx:=c
			”11” : mx:=d
selmy	3	I	”000”: my:=a
			”001”: my:=b
			”010”: my:=c
			”011”: my:=d
			”100”: my:=z1
selme	2	I	”00” : me:=0
			”01” : me:=d_{se}
			”10” : me:=d_{fp}
			”11” : me:=east
cta	1	I	’0’ : ma:=mm+me
			’1’ : ma:=mm-me
selmz	1	I	’0’ : z2:=ma
			’1’ : z2:=$z1_{se}$
selmb	2	I	”00” : mb:=0
			”01” : mb:=c_{se}
			”10” : mb:=c&d
			”11” : mb:=c_{fp}
selmo2	2	I	”00” : out2:=$o2_{fp}$
			”01” : out2:=$o2_L$
			”10” : out2:=$o2_H$
			”11” : out2:=$o1_H$
selmo1	2	I	”00” : out1:=$o1_{fp}$
			”01” : out1:=$o1_L$
			”10” : out1:=$o1_H$
			”11” : out1:=$o2_L$

Table A.2: ALU control signals (se=sign extend, fp=fixedpoint, L=lowest (least significant) part, H=highest (most significant) part), see Figure 2.2.

Appendix B

Configuration Registers

Bit	Register 1	Description
30..25	ctf3	alu select function 3
24..19	ctf2	alu select function 2
18..13	ctf1	alu select function 1
12..11	selmx	alu select mux x
10.. 8	selmy	alu select mux y
7	cta	alu select 0=add or 1=minus
6	selmz	alu select mux z
5.. 4	selmb	alu select mux b
3.. 2	selmo2	alu out2
1.. 0	selmo1	alu out1

Table B.1: Configuration registers 1, see Appendix A

Bit	Register 2	Description
31..28	rd_reg4	crossbar to Register 4 (Table 4.1, p. 36)
27..24	rd_reg3	crossbar to Register 3 (Table 4.1, p. 36)
23..20	rd_reg2	crossbar to Register 2 (Table 4.1, p. 36)
19..16	rd_reg1	crossbar to Register 1 (Table 4.1, p. 36)
15..13	wr_alu2	aluout2 to crossbar (Table 4.1, p. 36)
12..10	wr_alu1	aluout1 to crossbar (Table 4.1, p. 36)
9	aluout2_reg_addr_r	register 'aluout2' read address
8	aluout1_reg_addr_r	register 'aluout1' read address
7.. 6	reg2_addr_w	Register 2 write address
5.. 4	reg2_addr_r	Register 2 read address
3.. 2	reg1_addr_w	Register 1 write address
1.. 0	reg1_addr_r	Register 1 read address

Table B.2: Configuration registers 2

Bit	Register 3	Description
31	aluout2_reg_we	write enable for aluout register
30	aluout1_reg_we	write enable for aluout register
29	cnt2_en	Counter 2 enable (Section 4.1.4, p. 37)
28	cnt2_dir	Counter 2 direction (0=up, 1=down)
27	cnt2_byp	Counter 2 bypass (0=cnt, 1=reg)
26	cnt1_en	Counter 1 enable (Section 4.1.4, p. 37)
25	cnt1_dir	Counter 1 direction (0=up, 1=down)
24	cnt1_byp	Counter 1 bypass (0=cnt, 1=reg)
23..22	selme	alu select mux e (Appendix A)
21..20	reg4_addr_w	Register 4 write address
19..18	reg4_addr_r	Register 4 read address
17..16	reg3_addr_w	Register 3 write address
15..14	reg3_addr_r	Register 3 read address
13..11	wr_mem2	Memory 2 to crossbar (Table 4.1, p. 36)
10.. 8	wr_mem1	Memory 1 to crossbar (Table 4.1, p. 36)
7.. 4	rd_mem2	crossbar to Memory 2 (Table 4.1, p. 36)
3.. 0	rd_mem1	crossbar to Memory 1 (Table 4.1, p. 36)

Table B.3: Configuration registers 3

Bit	Reg. 4	Bit	Reg. 5	Description
31 .. 8	cnt1_reg_in	31..8	cnt2_reg_in	counter reg_in (Fig. 38)
7 .. 0	cnt1_din	7..0	cnt2_din	counter din (p. 38)

Table B.4: Configuration register 4 and 5

Appendix C

Macro for Simulation

C.1 Functional Simulation

```
quit -sim
vcom fpfa_types.vhd
vcom fpfa_utils_pkg.vhd
vcom fpfa_tristate.vhd
vcom fpfa_alu.vhd
vcom fpfa_reg_file.vhd
vcom fpfa_aluout_reg.vhd
vcom fpfa_crossbar_part.vhd
vcom fpfa_modelsim_mem.vhd
vcom fpfa_modelsim_mem16.vhd
vcom fpfa_cnt.vhd
vcom fpfa_config_reg.vhd
vcom fpfa_proc_part.vhd
vcom fpfa_tile_ctrl.vhd
vcom fpfa_tile.vhd
vcom fpfa_top.vhd
vcom fpfa_test_pkg.vhd
vcom fpfa_tb.vhd
vsim -t ps work.fpfa_tb
add wave -radix hexadecimal /fpfa_tb/*
run -all
```

C.2 Timing simulation

```
quit -sim
# vlib simprim
vmap simprim ./vhdl/simprim
# vcom simprim_Vpackage.vhd
# vcom simprim_VITAL.vhd
# vcom simprim_Vcomponents.vhd
vcom time_sim2.vhd
vcom fpfa_types.vhd
vcom fpfa_test_pkg.vhd
vcom fpfa_tb.vhd
vsim -t ps -sdftyp /fpfa_tb/top=time_sim2.sdf fpfa_tb
add wave -radix hexadecimal /fpfa_tb/*
run -all
```